(a) Multiply two complex numbers $(a + i\,b)$ and $(c + i\,d)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

(b) Find $(a + i\,b)^2$ using only two multiplications of real numbers.

(c) Find the product $(a + i\,b)^2(c + i\,d)^2$ using only five real number multiplications.

Solution:

(a) This is again the Karatsuba trick:

$$(a+i\,b)(c+i\,d) = ac-bd+(bc+ad)\,i = ac-bd+((a+b)(c+d)-ac-bd)\,i$$

so we need only 3 multiplications: ac and bd and $(a + b)(c + d)$.

(b) $(a + i\,b)^2 = a^2 - b^2 + 2ab\,i = (a + b)(a - b) + (a + a)b\,i.$

(c) Just note that $(a + i\,b)^2(c + i\,d)^2 = ((a + i\,b)(c + i\,d))^2$; you can now use (a) to multiply $(a + i\,b)(c + i\,d)$ with only three multiplications and then you can use (b) to square the result with two additional multiplications.

(a) Multiply two complex numbers $(a + i\,b)$ and $(c + i\,d)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

(b) Find $(a + i\,b)^2$ using only two multiplications of real numbers.

(c) Find the product $(a + i\,b)^2(c + i\,d)^2$ using only five real number multiplications.

Solution:

(a) This is again the Karatsuba trick:

$$(a+i\,b)(c+i\,d) = ac-bd+(bc+ad)\,i = ac-bd+((a+b)(c+d)-ac-bd)\,i$$

so we need only 3 multiplications: ac and bd and $(a + b)(c + d)$.

(b) $(a + i\,b)^2 = a^2 - b^2 + 2ab\,i = (a + b)(a - b) + (a + a)b\,i.$

(c) Just note that $(a + i\,b)^2(c + i\,d)^2 = ((a + i\,b)(c + i\,d))^2$; you can now use (a) to multiply $(a + i\,b)(c + i\,d)$ with only three multiplications and then you can use (b) to square the result with two additional multiplications.

(a) Multiply two complex numbers $(a + i\,b)$ and $(c + i\,d)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

(b) Find $(a + i\,b)^2$ using only two multiplications of real numbers.

(c) Find the product $(a + i\,b)^2(c + i\,d)^2$ using only five real number multiplications.

Solution:

(a) This is again the Karatsuba trick:

$$(a+i\,b)(c+i\,d) = ac-bd+(bc+ad)\,i = ac-bd+((a+b)(c+d)-ac-bd)\,i$$

so we need only 3 multiplications: ac and bd and $(a + b)(c + d)$.

(b) $(a + i\,b)^2 = a^2 - b^2 + 2ab\,i = (a + b)(a - b) + (a + a)b\,i.$

(c) Just note that $(a + i\,b)^2(c + i\,d)^2 = ((a + i\,b)(c + i\,d))^2$; you can now use (a) to multiply $(a + i\,b)(c + i\,d)$ with only three multiplications and then you can use (b) to square the result with two additional multiplications.

(a) Multiply two complex numbers $(a+i\,b)$ and $(c+i\,d)$ (where a,b,c,d are all real numbers) using only 3 real number multiplications.

(b) Find $(a+i\,b)^2$ using only two multiplications of real numbers.

(c) Find the product $(a+i\,b)^2(c+i\,d)^2$ using only five real number multiplications.

Solution:

(a) This is again the Karatsuba trick:

$$(a+i\,b)(c+i\,d)=ac-bd+(bc+ad)\,i=ac-bd+((a+b)(c+d)-ac-bd)\,i$$

so we need only 3 multiplications: ac and bd and $(a+b)(c+d)$.

(b) $(a+i\,b)^2=a^2-b^2+2ab\,i=(a+b)(a-b)+(a+a)b\,i$.

(c) Just note that $(a+i\,b)^2(c+i\,d)^2=((a+i\,b)(c+i\,d))^2$; you can now use (a) to multiply $(a+i\,b)(c+i\,d)$ with only three multiplications and then you can use (b) to square the result with two additional multiplications.

Let us define the Fibonacci numbers as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Thus, the Fibonacci sequence looks as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

(a) Show by induction that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) Give an algorithm that finds F_n in $O(\log n)$ time.

Solution

(a) When $n = 1$ we have

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1$$

so our claim is true for $n = 1$.

Let us define the Fibonacci numbers as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Thus, the Fibonacci sequence looks as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

(a) Show by induction that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) Give an algorithm that finds F_n in $O(\log n)$ time.

Solution

(a) When $n = 1$ we have

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \end{aligned}$$

so our claim is true for $n = 1$.

Let $k \geq 1$ be an integer, and suppose our claim holds for $n = k$ (Inductive Hypothesis). Then

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned}$$

by the Inductive Hypothesis. Hence

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} &= \begin{pmatrix} F_{k+1} + F_k & F_{k+1} \\ F_k + F_{k-1} & F_k \end{pmatrix} \\ &= \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix} \end{aligned}$$

by the definition of the Fibonacci numbers. Hence, our claim is also true for $n = k + 1$, so by induction it is true for all integers $n \geq 1$.

Let

$$G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- We can now proceed by divide and conquer.
- To compute G^n: if n is even, recursively compute $G^{n/2}$ and square it in $O(1)$.
- If n is odd, recursively compute $G^{(n-1)/2}$, square it and then multiply by another G: this last step also occurs in $O(1)$.
- Since there are $O(\log n)$ steps of the recursion only, this algorithm runs in $O(\log n)$.

Let

$$G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- We can now proceed by divide and conquer.
- To compute G^n: if n is even, recursively compute $G^{n/2}$ and square it in $O(1)$.
- If n is odd, recursively compute $G^{(n-1)/2}$, square it and then multiply by another G: this last step also occurs in $O(1)$.
- Since there are $O(\log n)$ steps of the recursion only, this algorithm runs in $O(\log n)$.

Let

$$G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- We can now proceed by divide and conquer.
- To compute G^n: if n is even, recursively compute $G^{n/2}$ and square it in $O(1)$.
- If n is odd, recursively compute $G^{(n-1)/2}$, square it and then multiply by another G: this last step also occurs in $O(1)$.
- Since there are $O(\log n)$ steps of the recursion only, this algorithm runs in $O(\log n)$.

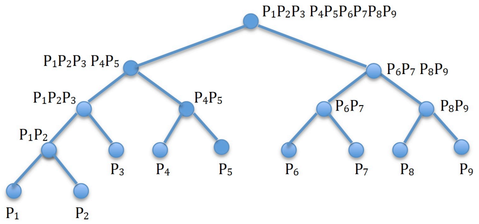
Let

$$G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

- We can now proceed by divide and conquer.
- To compute G^n: if n is even, recursively compute $G^{n/2}$ and square it in $O(1)$.
- If n is odd, recursively compute $G^{(n-1)/2}$, square it and then multiply by another G: this last step also occurs in $O(1)$.
- Since there are $O(\log n)$ steps of the recursion only, this algorithm runs in $O(\log n)$.

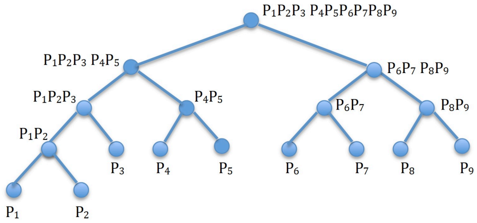
- Assume you are given n sorted arrays P_i, $1 \leq i \leq n$, of different sizes e_i. You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left. Design an algorithm that achieves this task and uses minimal total number of moves of elements of the arrays. Give an informal justification for why your algorithm is optimal.

  Solution: Clearly, the elements of the arrays that are merged earlier will be moved more times than the elements of the arrays that are merged later. Thus, we take the two shortest arrays and merge them, and continue with the resulting set of arrays in the same manner until there is only one array remaining.



- Assume you are given n sorted arrays P_i, $1 \leq i \leq n$, of different sizes e_i. You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left. Design an algorithm that achieves this task and uses minimal total number of moves of elements of the arrays. Give an informal justification for why your algorithm is optimal.

  Solution: Clearly, the elements of the arrays that are merged earlier will be moved more times than the elements of the arrays that are merged later. Thus, we take the two shortest arrays and merge them, and continue with the resulting set of arrays in the same manner until there is only one array remaining.



- You are given n disjoint closed intervals $X_1, \ldots, X_n$ of real numbers and an integer $k < n$. Construct a sequence of k intervals $Y_1, \ldots Y_k$ which cover intervals $X_1, \ldots, X_n$ and have a minimal total length. Thus, we should have $\bigcup_{i \leq n} X_i \subseteq \bigcup_{j \leq k} Y_j$.
  Solution 1: Cover all intervals $X_1, \ldots, X_n$ with a single interval Z_1. Then look for the largest gap and excise it from Z_1. Continue excising gaps between intervals $X_1, \ldots, X_n$ until you produce k intervals.
  Solution 2: Look for the shortest gap between two adjacent intervals X_i and X_{i+1} and cover them by a single interval Z_1. Continue joining nearest pairs of intervals till you end up with k many intervals.

- You are given n disjoint closed intervals $X_1, \ldots, X_n$ of real numbers and an integer $k < n$. Construct a sequence of k intervals $Y_1, \ldots Y_k$ which cover intervals $X_1, \ldots, X_n$ and have a minimal total length. Thus, we should have $\bigcup_{i \leq n} X_i \subseteq \bigcup_{j \leq k} Y_j$.
  Solution 1: Cover all intervals $X_1, \ldots, X_n$ with a single interval Z_1. Then look for the largest gap and excise it from Z_1. Continue excising gaps between intervals $X_1, \ldots, X_n$ until you produce k intervals.
  Solution 2: Look for the shortest gap between two adjacent intervals X_i and X_{i+1} and cover them by a single interval Z_1. Continue joining nearest pairs of intervals till you end up with k many intervals.

- You are given n disjoint closed intervals $X_1, \ldots, X_n$ of real numbers and an integer $k < n$. Construct a sequence of k intervals $Y_1, \ldots Y_k$ which cover intervals $X_1, \ldots, X_n$ and have a minimal total length. Thus, we should have $\bigcup_{i \leq n} X_i \subseteq \bigcup_{j \leq k} Y_j$.

  Solution 1: Cover all intervals $X_1, \ldots, X_n$ with a single interval Z_1. Then look for the largest gap and excise it from Z_1. Continue excising gaps between intervals $X_1, \ldots, X_n$ until you produce k intervals.

  Solution 2: Look for the shortest gap between two adjacent intervals X_i and X_{i+1} and cover them by a single interval Z_1. Continue joining nearest pairs of intervals till you end up with k many intervals.

- 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:

- 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, assume the opposite - that is, suppose that for a certain amount M, there is an optimal way of payment which is more efficient than the one described by the greedy algorithm.
- Clearly, 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.

- 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:
- 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, assume the opposite - that is, suppose that for a certain amount M, there is an optimal way of payment which is more efficient than the one described by the greedy algorithm.
- Clearly, 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.

- 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:
- 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, assume the opposite - that is, suppose that for a certain amount M, there is an optimal way of payment which is more efficient than the one described by the greedy algorithm.
- Clearly, 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.

- 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:

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

- This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar 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 dollar coin.
- Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins.
- For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins.
- Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins.
- Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin.

- This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar 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 dollar coin.
- Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins.
- For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins.
- Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins.
- Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin.

- This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar 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 dollar coin.
- Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins.
- For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins.
- Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins.
- Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin.

- This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar 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 dollar coin.
- Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins.
- For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins.
- Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins.
- Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin.

- This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 dollar 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 dollar coin.
- Thus, after the option of giving a \$1 dollar coin has been exhausted we are left with at least \$1 to be given without using any \$1 dollar coins.
- For the same reason at most one 50 cent coin can be given and we are left with an amount of at least 50 cents to be given without using any 50 cent coins.
- Note that at most two 20 cent coins can be used because three 20 cent coins can be replaced with a 50 cent and a 10 cent coins.
- Also note that if two 20 cent coins are used, no 10 cent coins can be used because two 20 cent coins and a 10 cent coin can be replaced with a single 50 cent coin.

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

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

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

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

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

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

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

- 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 which, given any amount, makes that amount using a minimal number of coins.

  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.

- 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 which, given any amount, makes that amount using a minimal number of coins.
  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.

- 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 which, given any amount, makes that amount using a minimal number of coins.
  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.

- 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 which, given any amount, makes that amount using a minimal number of coins.
  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.

- 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.

- 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.

- 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 $n, A, B, a[1 \ldots n]$ and $b[1 \ldots n]$, you have to determine the **maximum** total amount of money you can earn.

  Solution:

- Let $d[i] = |a[i] - b[i]|$; sort all $d[i]$ in a decreasing order and re-index all the items such that $|a[1] - b[1]|$ is the largest difference and so on, the i^{th} item being such that $d[i] = |a[i] - b[i]|$ is the i^{th} difference in size.
- We now go through the list giving the i^{th} 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.

- 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 $n, A, B, a[1 \ldots n]$ and $b[1 \ldots n]$, you have to determine the **maximum** total amount of money you can earn.
  Solution:
- Let $d[i] = |a[i] - b[i]|$; sort all $d[i]$ in a decreasing order and re-index all the items such that $|a[1] - b[1]|$ is the largest difference and so on, the i^{th} item being such that $d[i] = |a[i] - b[i]|$ is the i^{th} difference in size.
- We now go through the list giving the i^{th} 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.

- 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 $n, A, B, a[1 \ldots n]$ and $b[1 \ldots n]$, you have to determine the **maximum** total amount of money you can earn.

  Solution:

- Let $d[i] = |a[i] - b[i]|$; sort all $d[i]$ in a decreasing order and re-index all the items such that $|a[1] - b[1]|$ is the largest difference and so on, the i^{th} item being such that $d[i] = |a[i] - b[i]|$ is the i^{th} difference in size.
- We now go through the list giving the i^{th} 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.
- To see that this algorithm is optimal, let $m[i] = \min(a[i], b[i])$. Then regardless of who gets item i you get at least the amount $m[i]$, plus you get the amount $d[i]$ if item i is given to the higher bidder.
- Our algorithm tries to get as many $d[i]$'s as possible, preferentially taking as large $d[i]$'s as possible, so the algorithm is clearly optimal.
- 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 takes $O(n)$ time. Thus the whole algorithm runs in time $O(n \log n)$.

- 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.
- To see that this algorithm is optimal, let $m[i] = \min(a[i], b[i])$. Then regardless of who gets item i you get at least the amount $m[i]$, plus you get the amount $d[i]$ if item i is given to the higher bidder.
- Our algorithm tries to get as many $d[i]$'s as possible, preferentially taking as large $d[i]$'s as possible, so the algorithm is clearly optimal.
- 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 takes $O(n)$ time. Thus the whole algorithm runs in time $O(n \log n)$.

- 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.
- To see that this algorithm is optimal, let $m[i] = \min(a[i], b[i])$. Then regardless of who gets item i you get at least the amount $m[i]$, plus you get the amount $d[i]$ if item i is given to the higher bidder.
- Our algorithm tries to get as many $d[i]$'s as possible, preferentially taking as large $d[i]$'s as possible, so the algorithm is clearly optimal.
- 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 takes $O(n)$ time. Thus the whole algorithm runs in time $O(n \log n)$.

- 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.
- To see that this algorithm is optimal, let $m[i] = \min(a[i], b[i])$. Then regardless of who gets item i you get at least the amount $m[i]$, plus you get the amount $d[i]$ if item i is given to the higher bidder.
- Our algorithm tries to get as many $d[i]$'s as possible, preferentially taking as large $d[i]$'s as possible, so the algorithm is clearly optimal.
- 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 takes $O(n)$ time. Thus the whole algorithm runs in time $O(n \log n)$.

- 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:

- Compare running the Kruskal algorithm on old graph G and on the new graph G' with an additional vertex and corresponding edges.
- If an edge is not included in the minimal spanning tree of G because it would cause a cycle in G, clearly it cannot be included in a minimal spanning tree of G'.

- 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:

- Compare running the Kruskal algorithm on old graph G and on the new graph G' with an additional vertex and corresponding edges.
- If an edge is not included in the minimal spanning tree of G because it would cause a cycle in G, clearly it cannot be included in a minimal spanning tree of G'.

- 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:
- Compare running the Kruskal algorithm on old graph G and on the new graph G' with an additional vertex and corresponding edges.
- If an edge is not included in the minimal spanning tree of G because it would cause a cycle in G, clearly it cannot be included in a minimal spanning tree of G'.

- Thus, 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.
- So, 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 $O(n \log n)$.

- Thus, 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.
- So, 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 $O(n \log n)$.

- Thus, 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.
- So, 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 $O(n \log n)$.

- Assume that you are given a complete graph G with 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 $w(i,j)^2$.
    1. Assume that T is the minimal spanning tree of G. Does T necessarily remain the minimal spanning tree for the new graph G'?
    2. 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:

    1. Yes; just repeat Kruskal's algorithm with the new weights. The ordering of the edges does not change so the same minimum spanning tree will be produced.
    2. No; 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.

- Assume that you are given a complete graph G with 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 $w(i,j)^2$.
  1. Assume that T is the minimal spanning tree of G. Does T necessarily remain the minimal spanning tree for the new graph G'?
  2. 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:

1. Yes; just repeat Kruskal's algorithm with the new weights. The ordering of the edges does not change so the same minimum spanning tree will be produced.
2. No; 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.

- Assume that you are given a complete graph G with 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 $w(i,j)^2$.
  1. Assume that T is the minimal spanning tree of G. Does T necessarily remain the minimal spanning tree for the new graph G'?
  2. 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:

  1. Yes; just repeat Kruskal's algorithm with the new weights. The ordering of the edges does not change so the same minimum spanning tree will be produced.
  2. No; 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.