

**Due Friday 16<sup>th</sup> of June at 6pm Sydney time (week 3)**

In this assignment we review some basic algorithms and data structures, and we apply the divide-and-conquer paradigm. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of "one sympathy mark please" for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the [Ed forum](#). We will maintain a [FAQ thread](#) for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on [Overleaf](#), and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. ‘General’ here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is [UNSW Smarthinking](#), which is the university’s official writing support service.

Please review the UNSW policy on [plagiarism](#). Academic misconduct carries severe penalties.

Please read the [Frequently Asked Questions](#) document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

## Question 1

You are given an array  $A$  of  $n$  distinct positive integers and a positive integer  $x$ .

- 1.1 [8 marks]** Design an algorithm which decides if there exist two distinct indices  $1 \leq i, j \leq n$  such that  $2A[i] - 3A[j] = x$ . In the worst-case, your algorithm must run in  $O(n \log n)$  time.

We propose a binary search-based algorithm as follows.

**Algorithm:** Construct another array  $B[1 \dots n]$  such that  $B[j] = 3A[j] + x$  where  $j = 1, \dots, n$  and then sort  $B$  in ascending order using merge sort. Now we go through  $A$  and check for each  $i$  in  $1, \dots, n$  if  $2A[i]$  belongs to  $B$  using binary search. Any successful search indicates a yes to our question else it we return no.

**Correctness:** If there exists a some valid  $\alpha, \beta \in A$  then we must have  $2\alpha = 3\beta + x \in B$  which we can binary search for. In the case of no solution, the uniqueness of our solution guarantees that the binary search will return no result.

**Time complexity:** Merge sorting  $B$  takes  $O(n \log n)$ , then for each step of iteration through  $A$ , we use  $O(\log n)$  to binary search the value in  $B$  resulting in a total complexity of  $O(n \log n)$  again. Thus the whole algorithm runs in time  $O(n \log n)$ .

- 1.2 [4 marks]** Solve the same problem as in 1.1 but with an algorithm which runs in the **expected time** of  $O(n)$ .

Expected time complexity usually suggests that we use a *hash table*.

**Algorithm:** Go through  $A$  for each  $i = 1, \dots, n$  we insert  $3A[i] + x$  into our hash table  $H$ . Then we go through  $A$  again for  $j = 1, \dots, n$  and check if  $2A[j]$  can be found in the hash table. Similarly, any successful check indicates yes else we return no.

**Correctness:** Same as 1.1, the corresponding value  $2\alpha$  must be accessible in  $H$ .

**Time complexity:** For both iterations of  $A$ , we either hash or check if a value is in  $H$  which takes  $O(1)$  *expected time*. Therefore the total complexity is  $O(n)$  *expected time*.

- 1.3 [8 marks]** Design an algorithm which counts how many distinct pairs of indices  $(i, j)$  where  $1 \leq i < j \leq n$  satisfy both:

- $A[i] > A[j]$ ; and
- $A[i] + A[j] = x$ .

In the worst-case, your algorithm must run in  $O(n(\log n)^2)$  time.

We propose an solution that modifies the stand inversion counting algorithm.

**Algorithm:** When merging 2 sub-arrays in the standard inversion counting algorithm, let the sub-arrays that are being merged be  $A_{lo}[1 \dots k]$  and  $A_{hi}[1 \dots k]$ . Then during the process of finding the inversion, for each instance where  $A_{hi}[i] < A_{lo}[j]$ , we binary search and count how many times we find  $x - A_{hi}[i]$  within  $A_{lo}[j \dots k]$ . The occurrence we have counted is then our final solution.

**Correctness:** As both sub-arrays are sorted, we know that if  $A_{hi}[i] < A_{lo}[j]$  then all elements in  $A_{lo}[j \dots k]$  are part of an inversion with  $A_{hi}[i]$ . Then we only need to find if  $x - A_{hi}[i]$  is within  $A[j \dots k]$  which we can binary search as  $A_{lo}[1 \dots k]$  is sorted.

**Time complexity:** For each pair of inversion, we require an additional  $O(\log n)$  for binary

search, this changes our recurrence form to  $T(n) = 2T(n/2) + \Theta(n \log n)$  with  $f(n) = n \log n$  and the critical exponent  $c^* = \log 2 = 1$ . We see that as

- [A]  $f(n) \neq O(n^{1-\epsilon})$ ,
- [B]  $f(n) \neq \Theta(n)$ ,
- [C]  $f(n) \neq \Omega(n^{1+\epsilon})$ ,

all three cases of the master theorem do not apply. We can expand our recurrence via

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n \log n) = 2\left(T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2} \log \frac{n}{2}\right)\right) + \Theta(n \log n) \\ &= 2T\left(\frac{n}{4}\right) + 2\Theta\left(\frac{n}{2} \log \frac{n}{2}\right) + \Theta(n \log n) \end{aligned}$$

The pattern suggests for the depth of recursion being  $m = \log(n)$ ,

$$\begin{aligned} T(n) &= 2^m T\left(\frac{n}{2^m}\right) + \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k} \log \frac{n}{2^k}\right) \\ &= nT(1) + \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k} \log \frac{n}{2^k}\right). \end{aligned}$$

By looking the second term of the previous expression, we get

$$\begin{aligned} \sum_{k=0}^{m-1} 2^k \Theta\left(\frac{n}{2^k} \log \frac{n}{2^k}\right) &= \Theta\left(n \sum_{k=0}^{m-1} \log \frac{n}{2^k}\right) = \Theta\left(n \sum_{k=0}^{m-1} (\log n - \log 2^k)\right) \\ &= \Theta\left(n(\log n)^2 - \log n \log 2 \left(\sum_{k=0}^{m-1} k\right)\right) \\ &= \Theta\left(n(\log n)^2 - \log n \left(\frac{m(m-1)}{2}\right)\right) \\ &= \Theta(n(\log n)^2). \end{aligned}$$

**Alternative time complexity** We can also justify our complexity by recognizing that our recurrence forms a tree of height  $k = \lfloor \log n \rfloor$  and for each level, it takes  $O(n \log n)$  to compute all its merging (and searching) operations as the number of times we have to binary searches will not exceed the total length of  $A_{hi}[1 \dots k]$ . This hence gives us the final complexity of  $O(kn \log n) = O(n(\log n)^2)$ .

We can alternatively solve the problem in  $O(n \log n)$ .

### Algorithm

1. Form an array  $A' = [(A[1], 1), (A[2], 2), \dots, (A[n], n)]$ , then sorting with MERGESORT in ascending order of the first value of each element.
2. For each element  $i$  in  $A'$ , binary search for  $x - A[i]$ , and if found, check whether it is an inversion using the original index of the value.
3. Return all the distinct pairs found in step 2.

### Correctness

By creating and sorting the array  $A'$  that stores the index and value of each element in  $A$

we can use binary search to find if an element exists such that  $A[i] + A[j] = x$  for any  $i$  by searching for  $A[j] = x - A[i]$ . We can then check if it is an inversion in our original array by checking that original index (the second value stored in  $A'$ ). Repeating this process for all  $i$  in  $A'$  then allow us to find all distinct pairs that satisfy the required conditions.

### Time Complexity

In step 1, we generate a new array  $A'$  of size  $n$  based off of  $A$ , which takes  $O(n)$  time. Also in step 1, we then use MERGESORT on  $A'$ , which costs  $O(n \log n)$  time. In step 2 and 3, we execute a binary search on all  $n$  elements, so this takes  $O(n) * O(\log n) = O(n \log n)$  time. So in total, our algorithm runs in  $O(n) + O(n \log n) + O(n \log n) = O(n \log n)$  time as required.

## Question 2

**2.1 [6 marks]** Blake and Red each have a collection of  $n$  distinct Pokemon cards. Blake creates an array  $B$  containing the serial numbers of their cards, and Red creates an array  $R$  with the serial numbers of his. Blake wishes to know how many of their cards are *not* also owned by Red (that is, the size of  $B \setminus R$ ).

For example, if Blake has cards with serial numbers  $B = [7, 2, 1, 5]$  and Red has cards numbered  $R = [2, 8, 7, 3]$ , then there are 2 cards owned by Blake but not Red.

Design an algorithm which runs in  $O(n \log n)$  time and determines how many cards are owned by Blake but not Red.

**Algorithm:** We start by sorting  $R$  in an ascending order using merge sort. Then for each element  $B[i]$  for  $i = 1, \dots, n$ , we binary search the value of  $B[i]$  in  $R$ . The serial numbers of the desired cards are the values of  $B[i]$  that does not exist in  $R$ .

**Correctness:** For each element  $B[i]$  we iterate, we either have that  $B[i] \in B \setminus R$  where the binary search will return false or  $B[i] \in B \cap R$  where the binary search will return true. Both cases are covered by our solution.

**Time complexity:** For each element in  $B$  we expend  $O(\log n)$  to binary search its value in  $R$ , this results in a total complexity of  $O(n \log n)$ .

**2.2 [4 marks]** Gerald has a class of  $k$  students, each of which has a collection of distinct Pokemon cards. One day, he asks each member of his class to bring their collection, sorted by serial number. The  $i$ th student has  $c_i$  cards, with their **sorted** serial numbers in an array  $N_i[1 \dots c_i]$ . The total number of (not necessarily distinct) cards owned by the class is  $S = \sum_{i=1}^k c_i$ .

He wants to find all the distinct cards owned by the class (i.e.  $\bigcup_{i=1}^k N_i$ ), but needs to do so in  $O(S \log k)$  to finish the lesson in time. Gerald devises the following algorithm to do so:

We create an array  $N^*$  to store the serial numbers of the unique cards owned by the class. We first put all the numbers from  $N_1$  into  $N^*$ .

Then, for each  $i$  from 2 to  $k$ , we merge  $N_i$  with  $N^*$ , only including one copy of duplicate cards. Since  $N_i$  and  $N^*$  are both sorted, we are guaranteed to see duplicate cards one after the other while merging. We then replace  $N^*$  with the merged array.

The final array  $N^*$  contains the merged arrays  $N_1, N_2, \dots, N_k$  without duplicate serial numbers, i.e. all the distinct cards owned by the class.

This algorithm is correct, but unfortunately does not meet Gerald's  $O(S \log k)$  time complexity requirement. Justify why the worst-case time complexity of Gerald's algorithm is slower than  $O(S \log k)$ .

Note that an example with a fixed size (i.e., fixed  $S$  or  $k$ ) is **not** sufficient!

Suppose that all students have  $c_i = m$  for all  $i = 1 \dots k$  cards and all those cards are unique. Then merging the collection  $N_i$  takes  $im$  many computations (i.e.  $2m, 3m, \dots, km$ ) as each merge requires us to sequentially iterate through all cards that are already in  $N^*$ .

For the sum of an arithmetic progression, we have

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}.$$

Then our total computation is

$$S_m = \sum_{i=1}^k mi = m \sum_{i=1}^k i = \frac{mk(k+1)}{2} = \Theta(k^2m).$$

Then from the settings given by our question that  $m = S/k$ , we yield

$$\Theta(k^2m) = \Theta\left(\frac{k^2S}{k}\right) = \Theta(kS)$$

which is too slow.

**2.3 [10 marks]** Design an algorithm that finds the distinct cards owned by Gerald's class, and runs in  $O(S \log k)$  time.

**Algorithm:**

1. Recursively find all the distinct cards in the first  $k/2$  and the last  $k/2$  students.
2. Merge the results to determine the distinct cards by checking through each card one by one in both halves.
3. Base case: for  $n = 1$  students, the distinct cards are that of the one student

**Correctness:** The distinct cards of all the students are the distinct cards of the first half of the students and the second half of the students, so merging together our results maintains the validity. For a singular student, the distinct cards can only be the distinct cards of that singular student, so our base case is correct.

**Time Complexity:** At each layer in merging, each of the remaining cards are checked once, which has an  $O(S)$  cost. Further, we half the number of sub-problems to solve at each step, so for  $k$  students, this will take  $O(\log k)$  time. Thus our algorithm runs in  $O(S \log k)$  time as required.

### Question 3

**3.1** Read about the asymptotic notation in the review material and determine if  $f(n) = O(g(n))$  or  $g(n) = O(f(n))$  or both (i.e.,  $f(n) = \Theta(g(n))$ ) or neither of the two, for the following pairs of functions.

[A] [5 marks]

$$f(n) = \log_2(n); \quad g(n) = \sqrt[5]{n}$$

[B] [5 marks]

$$f(n) = n^n; \quad g(n) = 2^{n \log_2(n^2)}$$

[C] [5 marks]

$$f(n) = n^{1+\cos(\pi n)}; \quad g(n) = n$$

You might find L'Hôpital's rule useful: if  $f(x), g(x) \rightarrow \infty$  as  $x \rightarrow \infty$  and they are differentiable, then

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

[A] We see that both  $f(n)$  and  $g(n)$  are differentiable. Also, as  $n \rightarrow \infty$ ,

$$f(n) = \log_2(n) \rightarrow \infty \quad \text{and} \quad g(n) = \sqrt[5]{n} \rightarrow \infty.$$

Then we can use L'Hôpital's rule to check the ratio between  $f(n)$  and  $g(n)$  to see which function dominates as  $n \rightarrow \infty$ . Expanding our expression yields

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt[5]{n}} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n \ln 2}}{\frac{1}{5n^{4/5}}} = \lim_{n \rightarrow \infty} \frac{5}{n^{1/5} \ln 2} = 0$$

Thus,  $\log_2 n = O(\sqrt[5]{n})$ .

[B] We see that as  $g(n)$  is 2 raised to power of multiples of  $\log_2(n)$ , we can manipulate and get

$$2^{n \log_2(n^2)} = 2^{2n \log_2(n)} = (2^{\log n})^{2n} = (n^n)^2 = f(n)^2.$$

Thus it is clear that  $n^n = O(2^{n \log_2(n^2)})$ .

[C] Note that for a periodic function like  $\cos(n)$ , we always have for all odd values of  $n$  we get

$$n^{1+\cos(\pi n)} = n^{1-1} = n^0 = 1$$

and for all even values of  $n$  we get

$$n^{1+\cos(\pi n)} = n^{1+1} = n^2.$$

Thus neither  $f(n) = O(g(n))$  nor  $g(n) = O(f(n))$  as  $f(n)$  oscillates infinitely around (above and below)  $g(n)$  as  $n \rightarrow \infty$ .

**3.2** [5 marks] Given  $n$  strings, each of length at most  $m$ , a divide and conquer approach can be used to find the longest common prefix amongst the strings. For example, the longest common prefix amongst `apple`, `apply` and `apart` is `ap`. Song has provided an algorithm to do so below:

1. Recursively determine the longest common prefixes among the first  $n/2$  words and the last  $n/2$  words.
2. Merge the results to determine the longest common prefix between the two halves by scanning through character by character.
3. Base case: for  $n = 1$ , the longest common prefix is the entire string.

However, Song isn't sure about the time complexity of the above algorithm. In Big-Theta notation, explain and justify what the time complexity of the above algorithm is.

Without loss of generality we assume that all  $n$  strings have a length of  $m$ , then the step of our algorithm will have a time complexity of  $\Theta(m)$ . This gives us a recurrence form of  $T(n) = 2T(n/2) + \Theta(m)$  and a recursion depth of  $\lceil \log_2(n) \rceil$ . At each level of the recursion, we have 2 times the amount of instances of the previous level with depth  $\lceil \log_2(n) \rceil$  having  $n$  instances.

From the sum of a geometric progression, we have

$$\sum_{k=1}^{\infty} \frac{1}{2^k} = 1.$$

We can then count the total number of instances of our recursion across all depths as

$$D = 1 + 2 + 4 + \dots + \frac{n}{4} + \frac{n}{2} = \sum_{k=1}^{\lceil \log_2(n) \rceil} \frac{n}{2^k} = n \left( \sum_{k=1}^{\lceil \log_2(n) \rceil} \frac{1}{2^k} \right) < n.$$

As each instance requires  $\Theta(m)$  time to compute the longest common prefix, our final complexity is

$$\Theta(Dm) = \Theta(nm).$$

**Due Monday 10<sup>th</sup> of July at 6pm Sydney time (week 7)**

In this assignment we review some basic algorithms and data structures, and we apply the greedy method. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of “one sympathy mark please” for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the [Ed forum](#). We will maintain a [FAQ thread](#) for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on [Overleaf](#), and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. ‘General’ here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is [UNSW Smarthinking](#), which is the university’s official writing support service.

Please review the UNSW policy on [plagiarism](#). Academic misconduct carries severe penalties.

Please read the [Frequently Asked Questions](#) document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

## Question 1

**1.1 [12 marks]** You are preparing to cross the Grand Canyon on a tightrope, and are building your balance pole out of steel rods. Naturally, you want to do this as cheaply as possible.

You have made a trip to Bunnings, and picked up  $n$  rods, the  $i$ th of which has length  $\ell_i$ . You need to weld these all together to create your balance pole. Welding rods  $x$  and  $y$  costs  $\ell_x + \ell_y$  dollars, and results in a new rod with length  $\ell_x + \ell_y$ .

For example, if we have rods with lengths [2, 1, 3]:

- After welding rods 2 and 3, costing \$4, we have rods with lengths [2, 4]
- We weld the remaining two rods, costing \$6
- The total cost to create the pole is \$10

Note that this is not necessarily the optimal sequence for these lengths.

Design an  $O(n \log n)$  algorithm which finds the order you should weld the rods to minimize the total cost of welding.

This problem is solved using a method similar to the construction of the Huffman codes.

**Algorithm:** The core idea is to always take the two shortest rods and weld them together, producing a new rod, to minimise the number of times the larger rods are included in the cost. This can be done in several ways such as adding  $\ell_i$  for  $i = 1, \dots, n$  to a min-heap  $Q$ , and repeatedly popping the head of  $Q$  twice with values  $\ell_i, \ell_j$  which will be the 2 rods being welded together. Then we push the value  $\ell_i + \ell_j$  back onto  $Q$  and add  $\ell_i + \ell_j$  to the total cost. We repeat this until  $Q$  has only one element.

**Time complexity:** Each step, we perform two pops and one push on  $Q$ , each of which has a time complexity of  $\log n$ . This reduces the size of  $Q$  by one each step, and we finish when the size is 1, so the algorithm runs for  $n$  steps, giving a time complexity of  $O(n \log n)$ .

**Correctness:** The proof of correctness follows very closely to the proof of correctness of Huffman coding. The key observation is that, for a particular rod  $x$ , the number of times it is used in the construction is *exactly* its depth in the construction of the heap (from here, we will just refer to it as a *tree*). Let  $T$  be the labelled binary tree constructed by our greedy algorithm and  $d_T(i)$  denote the depth of rod  $i$  in the tree  $T$ .

To prove correctness via the exchange argument, we need to prove the following claim:

**Claim.** *Let  $x$  and  $y$  be two rods with the smallest lengths, then there exist an optimal ordering where  $x$  and  $y$  are used the same number of times.*

*Proof.* Without the loss of generality, we may assume that  $\ell_x \leq \ell_y$ .

Observe that this is exactly the same as claiming that  $x$  and  $y$  have the same depth in any tree representing an optimal ordering. Therefore, we will prove that a given optimal tree can be transformed into a tree where  $x$  and  $y$  have the same depth. To do this, observe that the cost of a tree  $T$  is exactly

$$C(T) = \sum_{i=1}^n \ell_i \cdot d_T(i). \quad (1)$$

Let  $T$  be any tree representing an optimal ordering, and let rods  $a$  and  $b$  be the two rods of maximum depth in  $T$ . Without the loss of generality, assume that  $\ell_a \leq \ell_b$ . Since  $x$  and  $y$

have the smallest lengths, we have that

$$\ell_x \leq \ell_a, \quad \ell_y \leq \ell_b.$$

Now, if  $\ell_x = \ell_b$ , then necessarily, we have that  $\ell_x = \ell_b = \ell_a = \ell_y$  and there is nothing to prove. Therefore, we may assume that  $\ell_x \neq \ell_b$  and so,  $x \neq b$ . We will now exchange  $x$  with  $a$  to obtain a new tree  $T'$ . From (1), we see that

$$\begin{aligned} C(T) - C(T') &= \sum_{i=1}^n \ell_i \cdot d_T(i) - \sum_{i=1}^n \ell_i \cdot d_{T'}(i) \\ &= (\ell_a \cdot d_T(a) + \ell_x \cdot d_T(x)) - (\ell_a \cdot d_{T'}(a) + \ell_x \cdot d_{T'}(x)) \\ &= (\ell_a \cdot d_T(a) + \ell_x \cdot d_T(x)) - (\ell_a \cdot d_T(x) + \ell_x \cdot d_T(a)) \\ &= (d_T(a) - d_T(x))(\ell_a - \ell_x). \end{aligned}$$

Now, in tree  $T$ , rod  $a$  has maximum depth; therefore,  $d_T(a) \geq d_T(x)$  which implies that  $d_T(a) - d_T(x) \geq 0$ . Since  $\ell_x \leq \ell_a$ , we have that  $\ell_a - \ell_x \geq 0$ . In other words, we see that  $C(T) - C(T') \geq 0$  or equivalently,  $C(T) \geq C(T')$ . Thus, if tree  $T$  represents any optimal ordering, then necessarily,  $C(T) \leq C(T')$  for any tree  $T'$ . But this implies that  $C(T) = C(T')$  and we deduce that the ordering represented by  $T'$  is optimal. We can also exchange  $y$  with  $b$  in  $T'$  to generate our greedy tree  $T''$  and the argument follows. Thus, we have transformed any optimal tree  $T$  into  $T''$  while maintaining optimality. Therefore, there exist an optimal ordering where  $x$  and  $y$  are used the same number of times.  $\square$

To conclude that our greedy strategy is optimal, we inductively apply the exchange argument.

*Proof.* Let  $\mathcal{G} = (g_1, \dots, g_m)$  denote the our greedy ordering of the  $n$  rods and let  $\mathcal{O} = (o_1, \dots, o_{m'})$  denote the ordering from any optimal solution. Additionally, let  $g_i$  denote the cost of our greedy welding after the first  $i$  welds and let  $o_i$  denote the cost of  $\mathcal{O}$ 's welding after the first  $i$  welds. Clearly, if  $n = 1$ , any ordering is trivially optimal; therefore, assume that  $n \geq 2$ . We begin the induction proof.

- **Base case:** Consider the smallest two rods. By the claim, the cost must be at most the cost of any optimal solution after the first weld; therefore,  $g_1 \leq o_1$ .
- **Inductive step:** Assume that, after the first  $k$  welds,  $g_k \leq o_k$ . Again, apply the claim. Welding the smallest two rods produces a tree whose cost  $C$  is minimal; in other words, for any other pair of rods to weld with cost  $C'$ , we have that  $C \leq C'$ . Therefore, we have that  $g_{k+1} = g_k + C \leq o_k + C' = o_{k+1}$ , which proves the inductive step.

This proves that our greedy solution is correct and by the exchange argument, any solution that deviates from our greedy ordering is no better than our solution.  $\square$

**1.2 [8 marks]** Just before you step onto the wire, your pole snaps in half, and you realise going for the cheapest option might not have been the best idea. Fortunately, you brought spare rods and a welding device to the canyon.

As in part 1, you have  $n$  rods, the  $i$ th of which has length  $\ell_i$ . It is given that  $n$  is odd.

Each rod in the pole contributes an *instability*, which is the product of the rod's length and the number of rods it is away from the middle. The order in which the welds are performed does not matter, only the order of the rods in the final pole.

That is, if you weld the  $n$  rods to create a pole  $r_1, r_2, \dots, r_n$ , then the instability of the pole is

given by

$$\sum_{i=1}^n r_i \times \left| i - \frac{n+1}{2} \right|.$$

For example, if we create a pole [2, 4, 6, 5, 2], then it has instability

$$\begin{aligned} &= (2 \times |1 - 3|) + (4 \times |2 - 3|) + (6 \times |3 - 3|) + (5 \times |4 - 3|) + (2 \times |5 - 3|) \\ &= (2 \times 2) + (4 \times 1) + (6 \times 0) + (5 \times 1) + (2 \times 2) \\ &= 4 + 4 + 0 + 5 + 4 \\ &= 17. \end{aligned}$$

Design an  $O(n \log n)$  algorithm which finds the order of rods in the pole with minimal instability.

The main premise in this solution is to put the longest rods in the middle of the combined weld so they have minimal impact on the total instability of the combined rod.

#### Algorithm:

1. Sort the rods in descending order of length, using mergesort.
2. Take the longest rod, and attach the next two longest rods to the left and right of it.
3. Take the next two longest rods, and attach them to the left and right of the combined rods
4. Repeat step 3 until there are no more rods remaining

**Time Complexity:** Sorting the  $n$  rods with mergesort will take a total of  $O(n \log n)$  time. Then, we go through the sorted rods and add them one by one. Adding each rod will take  $O(1)$  time, for a total of  $O(n)$  rods. So the total runtime of this algorithm is  $O(n \log n) + O(n) = O(n \log n)$ .

**Correctness:** First note that this is extremely similar to the tape storage problem in the lecture slides. This is because we are essentially given a set of multipliers to each of the rods lengths, and asked to minimise this - the multipliers in being  $\left[0, 1, 1, 2, 2, \dots, \frac{(n-1)}{2}, \frac{(n-1)}{2}\right]$ . Our algorithm places the most expensive weights in the middle, i.e.: with lesser multipliers, so we have that our algorithm will produce a total instability of

$$(0 \times l_1) + (1 \times l_2) + (1 \times l_3) + \dots + \left( \left( \frac{(n-1)}{2} \times l_{n-1} \right) + \left( \frac{(n-1)}{2} \times l_n \right) \right),$$

where  $l_1 \geq l_2 \geq \dots \geq l_n$ . In other words, we are allocating the greatest length rods with the least multipliers available.

First let us define an inversion in this context. An allocation  $\delta$  has an inversion if it allocates a rod  $i$  of length  $l_i$  a multiplier of  $m_i$  and a rod  $j$  of length  $l_j$  a multiplier of  $m_j$ , where  $m_i > m_j$  and  $l_i > l_j$ .

**Claim.** *Our greedy strategy is correct*

Now let us show that this allocation of lengths to multipliers is optimal. Consider any arbitrary solution  $\delta$  that violates our greedy allocation by at least one pair of rods. For the first pair of rods that differ between our greedy solution  $G$  and  $\delta$ , let the rods have denoted length  $l_i$  and  $l_j$ , where  $l_i > l_j$ , and  $G$  have allocated the multipliers  $m_1$  to  $l_i$  and  $m_2$  to  $l_i$ , where  $m_1 > m_2$ . Our greedy solution would have chosen to allocate the  $i$  the multiplier  $m_2$ , and the rod  $j$  the multiplier  $m_1$ . Since  $m_1 > m_2$  and  $l_i > l_j$ , this clearly decreases the overall instability of the combined rods, and swapping any two arbitrary rods has no impact on the

instability of the others, as their multiplier is dependent on the number of rods it is away from the middle, not distance from the middle.

We simply continue to swap pairs of rods  $i$  and  $j$  that form an inversion, until there are no more inversions left, increasing optimally at each swap, until we eventually reach our greedy solution. Therefore our greedy schedule must be as optimal as any other allocation with no inversions, so our greedy solution is, indeed, optimal.

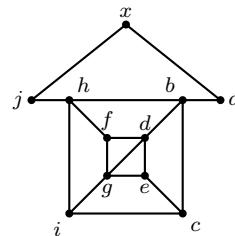
## Question 2

Sam has recently acquired Friendbook, a large social media conglomerate, under dubious circumstances. Since their position as the company's CEO is only probational, they wish to curry favour with the employees by providing raises. However, they don't want to give raises to those who aren't well-known in the company (that would be a waste), and they can't give raises to those who are too well-known (that would raise suspicions). As such, they want to figure out the largest number of people in the company they can give a raise to, without wasting money, and without being caught.

Thankfully for Sam, the company keeps an absurd amount of data on its employees, so they have access to an undirected graph  $G$  of all  $n$  company employees and who they know within the company. In particular, this graph is provided as an **adjacency matrix**  $M$  such that  $M[i, j] = 1$  if and only if employee  $i$  and  $j$  know each other. An **induced subgraph**  $H$  of  $G$  is a graph formed from a subset of the vertices of  $G$ , where pairs of vertices in  $H$  are adjacent if and only if the corresponding vertices are adjacent in  $G$ . In other words, an induced subgraph is formed by deleting some (or maybe none) of the vertices of  $G$ , and all adjacent edges.

Sam wishes to find the largest possible induced subgraph  $H$  of  $G$  such that every vertex in  $H$  is adjacent to at least  $k$  other vertices of  $H$ , but is also **not** adjacent to at least  $k$  other vertices of  $H$ . Such a subgraph is called “maximally nepotistic”.

**2.1 [2 marks]** Consider the following graph  $G$  of company connections:



With  $k = 3$ , explain why any maximally nepotistic subgraph of  $G$  **cannot** include the vertex labelled  $x$ .

It is connected to only two vertices in  $G$ , so no matter what subgraph we choose including it, it cannot be connected to at least  $k = 3$  vertices.

**2.2 [2 marks]** Using the graph  $G$  defined in 2.1, and again with  $k = 3$ , find a maximally nepotistic subgraph  $H$  of  $G$ , and explicitly list the vertices of  $H$ .

The set of vertices is  $\{b, c, d, e, f, g, h, i\}$ .

**2.3 [16 marks]** Given a graph  $G$  with  $n$  vertices and adjacency matrix  $M$ , and integer  $k \geq 1$ , design an algorithm which finds a maximally nepotistic subgraph of  $G$ , if it exists, and lists its vertices. If no such subgraph exists, the algorithm should not output anything. Your algorithm must run in time  $O(n^3)$ .

Consider deleting a vertex  $v \in V$  and all of its incident edges. For each vertex  $u$  connected to  $v$  (i.e.  $M[u, v] = 1$ ), the degree of  $u$  must necessarily decrease by 1. On the other hand, for each vertex  $w$  *not* connected to  $v$  (i.e.  $M[w, v] = 0$ ), the degree remains unchanged. In either case, deleting a vertex does not increase the degree of any vertex and can only decrease by at most 1.

**Algorithm:** With this observation, we can start determining what vertices are *never* included in a *maximally nepotistic* subgraph. Clearly, any vertex that is either adjacent to fewer than  $k$  vertices in  $G$  or not adjacent to fewer than  $k$  vertices in  $G$  can never belong in such a subgraph; therefore, we can safely remove these vertices and the corresponding incident edges. We repeat this on the subgraph induced by the remaining vertices, deleting vertices that are either connected to fewer than  $k$  vertices or not connected to fewer than  $k$  vertices until we've removed all of the vertices of  $G$  or we've obtained an induced subgraph  $H$  such that no vertex can be deleted anymore. If such a subgraph exists, it must necessarily be *nepotistic*. We will prove that it is maximal.

**Correctness:** Let  $H$  be the nepotistic subgraph generated by our greedy algorithm, and let  $H'$  be a nepotistic subgraph such that  $H$  is a subgraph of  $H'$ . Consider the set of vertices in  $H'$  but not  $H$ . We shall prove that such a set must be empty. Suppose otherwise. Let  $v$  be such a vertex. Moreover, since the greedy algorithm deletes vertices from  $G$ , it must have arrived at graphs that contain  $H$  as a subgraph. In particular, the greedy algorithm must arrive at  $H'$  before  $H$ . Since  $H'$  is nepotistic, this implies that  $v$  is adjacent to at least  $k$  vertices in  $H'$  and not adjacent to at least  $K$  vertices in  $H'$ . This implies that  $v$  belongs in the subgraph that is returned by the algorithm; in other words,  $v \in H$ , which is a contradiction. Therefore, the set of vertices in  $H'$  but not  $H$  must be empty, which implies that  $H' = H$ . This shows that  $H$  is maximal, proving that our greedy algorithm returns a maximal nepotistic subgraph.

**Time complexity:** We now examine the running time of the algorithm. To find a vertex to delete, we traverse through each row and count the number of 0s and 1s in the adjacency matrix. This has running time  $O(n)$  since we have to check at most  $n$  vertices. To ensure that we do not reconsider any deleted vertex  $i$ , we can remove row  $i$  and column  $i$  by either blocking out the row and column, or replacing the entries with a value that is different to 0 or 1. Therefore, finding all vertices to delete in one iteration takes  $O(n^2)$  time. Since each iteration removes a subset of vertices, we have *at most*  $n$  iterations of the algorithm, giving us a running time of  $O(n^3)$ .

### Question 3

**3.1 [10 marks]** Santa's assistant has been laid off and it is now your job to ensure that as many **wrapped** presents get delivered to shopping centres as possible.

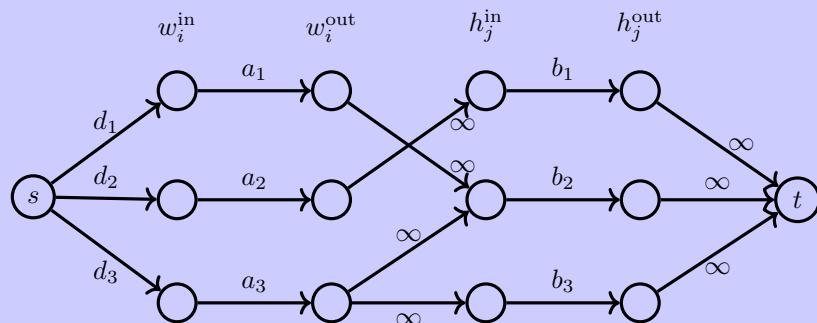
Santa has an infinite supply of **unwrapped** presents at the North Pole, and can distribute them to  $k$  workshops across the globe. At each workshop  $i$ , a maximum of  $a_i$  presents can be wrapped, and the North Pole can distribute to it at most  $d_i$  presents.

The workshops wrap the presents, and then deliver these to  $m$  shopping centres. Each shopping centre  $j$  can hold up to  $b_j$  packages, however, due to some legacy contractual agreements, each shopping centre can only receive presents from certain workshops. You are given the contractual agreements as a two dimensional array  $C$ , where  $C[i][j]$  denotes that workshop  $i$  has an agreement with shopping centre  $j$ .

Design an algorithm that runs in  $O((k + m)k^2m^2)$  time which finds the maximum number of wrapped presents that can be delivered to shopping centres.

**Graph construction:** We construct a flow network as follows.

- Construct a source vertex  $s$  that represents the North Pole.
- For each workshop  $i$ , construct two vertices,  $w_i^{\text{in}}$  and  $w_i^{\text{out}}$ .
- For each shopping centre  $j$ , construct two vertices,  $h_j^{\text{in}}$  and  $h_j^{\text{out}}$ .
- Construct a sink vertex  $t$ .
- For each workshop  $w_i$ , construct an edge from  $s$  to  $w_i^{\text{in}}$  with capacity  $d_i$ , representing the maximum amount of presents that the North Pole can send to workshop  $w_i$ .
- For each workshop  $i$ , construct an edge from  $w_i^{\text{in}}$  to  $w_i^{\text{out}}$  with capacity  $a_i$ , representing the maximum amount of presents that can be wrapped at workshop  $i$ .
- For each workshop  $i$  and shopping centre  $j$ , construct an edge from  $w_i^{\text{out}}$  to  $h_j^{\text{in}}$  of infinite capacity if and only if  $C[i][j] = \text{true}$ .
- For each shopping centre  $j$ , construct an edge from  $h_j^{\text{in}}$  to  $h_j^{\text{out}}$  with capacity  $b_j$ , representing the maximum amount of presents that a shopping centre can hold.
- For each shopping centre  $j$ , construct an edge from  $h_j^{\text{out}}$  to  $t$  of infinite capacity.



A flow network with  $k = 3$  workshops and  $m = 3$  shopping centres.

**Algorithm and solution:** We now run Edmonds-Karp on the flow network. Each unit of flow corresponds to one present being delivered from the North Pole to the shopping centres. The flow is bottlenecked by the amount of presents in the workshops and shopping centres.

In particular, each capacity from  $w_i^{\text{in}}$  to  $w_i^{\text{out}}$  enforces at most  $a_i$  presents in workshop  $i$ , the capacity from  $h_j^{\text{in}}$  to  $h_j^{\text{out}}$  enforces at most  $b_j$  presents in shopping centre  $j$ . Therefore, the maximum number of presents delivered from the North Pole to the shopping centres is exactly the maximum flow.

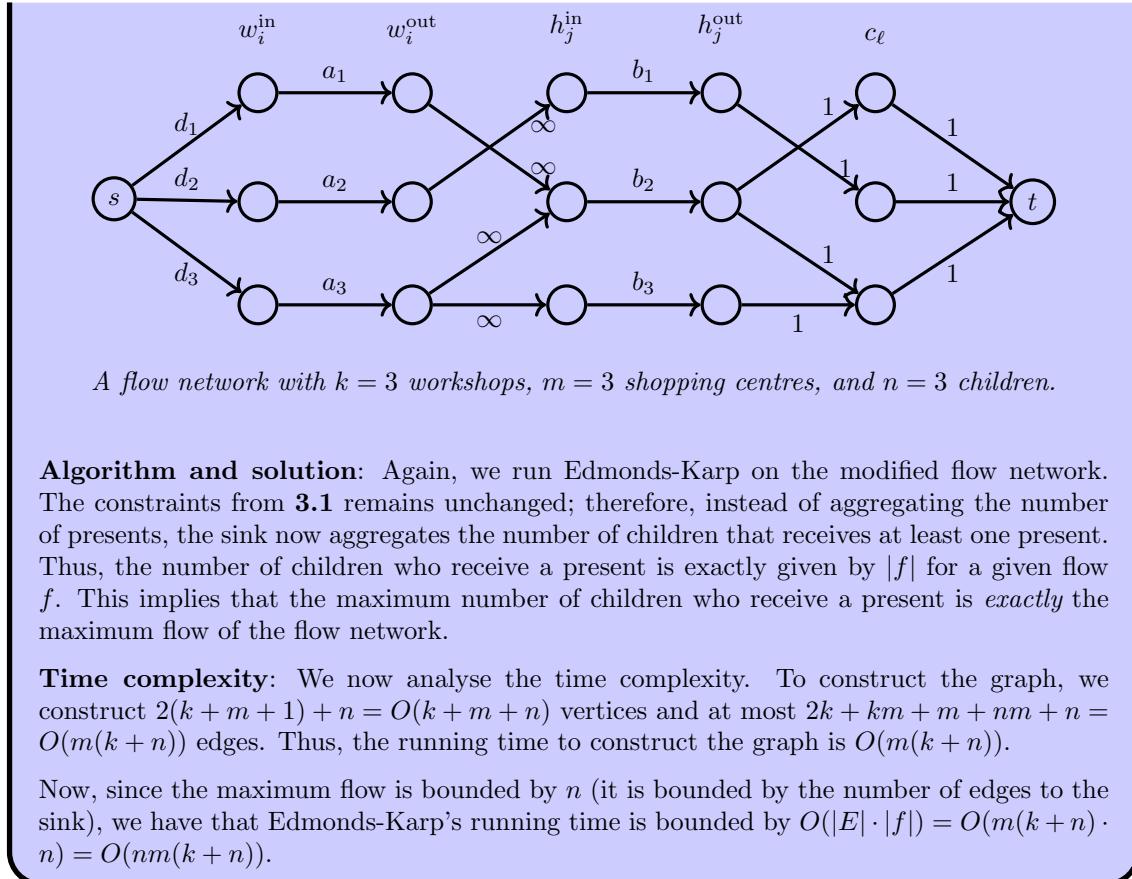
**Time complexity:** We now analyse the time complexity. To construct the graph, we construct  $2(k + m + 1) = O(k + m)$  vertices and at most  $2(k + m) + km = O(km)$  edges. Therefore, the construction of the graph has running time  $O(km)$ . The running time of Edmonds-Karp is  $O(|V| \cdot |E|^2) = O((k + m)(km)^2) = O((k + m)k^2m^2)$ . Therefore, the overall time complexity is  $O((k + m)k^2m^2)$ .

**3.2 [6 marks]** On Christmas Eve, you realise that you should ensure that each of the  $n$  children on Santa's nice list receive at least one present. Children are willing to travel at most  $D$  distance to a shopping centre to receive a present. You are given the location of all children and shopping centres.

Design an algorithm that runs in  $O(nm(k + n))$  time which finds the maximum number of children on the nice list who can receive at least one present.

**Graph construction:** We construct a flow network similar to **3.1** with a slight modification, as follows. Let  $\text{dist}(\ell, j)$  denote the distance from child  $\ell$  to shopping centre  $j$ .

- Construct a source vertex  $s$  that represents the North Pole.
- For each workshop  $i$ , construct two vertices,  $w_i^{\text{in}}$  and  $w_i^{\text{out}}$ .
- For each shopping centre  $j$ , construct two vertices,  $h_j^{\text{in}}$  and  $h_j^{\text{out}}$ .
- For each child  $\ell$ , construct a vertex  $c_\ell$ .
- Construct a sink vertex  $t$ .
- For each workshop  $w_i$ , construct an edge from  $s$  to  $w_i^{\text{in}}$  with capacity  $d_i$ , representing the maximum amount of presents that the North Pole can send to workshop  $w_i$ .
- For each workshop  $i$ , construct an edge from  $w_i^{\text{in}}$  to  $w_i^{\text{out}}$  with capacity  $a_i$ , representing the maximum amount of presents that can be wrapped at workshop  $i$ .
- For each workshop  $i$  and shopping centre  $j$ , construct an edge from  $w_i^{\text{out}}$  to  $h_j^{\text{in}}$  of infinite capacity if and only if  $C[i][j] = \text{true}$ .
- For each shopping centre  $j$ , construct an edge from  $h_j^{\text{in}}$  to  $h_j^{\text{out}}$  with capacity  $b_j$ , representing the maximum amount of presents that a shopping centre can hold.
- For each child  $\ell$  and shopping centre  $j$ , construct an edge from  $h_j^{\text{out}}$  to  $c_\ell$  with capacity 1 if and only if  $\text{dist}(\ell, j) \leq D$ , representing that child  $\ell$  is willing to travel to shopping centre  $j$ .
- For each child  $\ell$ , construct an edge from  $c_\ell$  to  $t$  with capacity 1.



**Algorithm and solution:** Again, we run Edmonds-Karp on the modified flow network. The constraints from 3.1 remains unchanged; therefore, instead of aggregating the number of presents, the sink now aggregates the number of children that receives at least one present. Thus, the number of children who receive a present is exactly given by  $|f|$  for a given flow  $f$ . This implies that the maximum number of children who receive a present is *exactly* the maximum flow of the flow network.

**Time complexity:** We now analyse the time complexity. To construct the graph, we construct  $2(k + m + 1) + n = O(k + m + n)$  vertices and at most  $2k + km + m + nm + n = O(m(k + n))$  edges. Thus, the running time to construct the graph is  $O(m(k + n))$ .

Now, since the maximum flow is bounded by  $n$  (it is bounded by the number of edges to the sink), we have that Edmonds-Karp's running time is bounded by  $O(|E| \cdot |f|) = O(m(k + n) \cdot n) = O(nm(k + n))$ .

### 3.3 [4 marks] Let the farthest distance from any child to a shopping centre be $F$ .

Design an algorithm that runs in  $O(nm(k + n) \log F)$  time which finds the minimum **integer** value of  $D$  such that all  $n$  children on the nice list receive at least one present, or returns that no such  $D$  is possible.

**Algorithm:** For each distance  $D \in \{1, \dots, F\}$ , perform the algorithm from 3.2 and check whether the maximum flow is equal to  $n$ .

- If the maximum flow is equal to  $n$ , then we conclude that it is possible for all  $n$  children on the nice list to receive at least one present. In this case, we recurse on the left subarray (including  $D$ ) and check for distances smaller than  $D$  by performing a binary search.
- If the maximum flow is smaller than  $n$ , then we conclude that it is not possible for all  $n$  children to receive at least one present. In this case, we recurse on the right subarray and check for distances larger than  $D$  by performing a binary search.

The algorithm then returns the smallest distance such that the maximum flow is equal to  $n$ .

**Correctness:** We first prove that binary search is applicable in this setting. In 3.2, we described an algorithm that finds the maximum number (i.e. the maximum flow) given a distance  $D$ . Let  $S$  denote the set of children who received a present given distance  $D$  and let  $|f_D|$  denote the maximum flow with distance  $D$ .

- Since the dist function is monotone, increasing the distance to  $D' \geq D$  increases the number of edges from  $h_j^{\text{out}} \rightarrow c_k$ . Since the same children from  $S$  remain unaffected

when we increase the distance to  $D'$ , the maximum flow given distance  $D'$  must be at least the maximum flow given distance  $D$  (i.e.  $|f_D| \leq |f_{D'}|$ ). However, since  $|f_D| \leq n$  for all  $D$ , this implies that, if it was possible with distance  $D$ , it must also be possible with distance  $D'$ .

- We now prove the reverse. Suppose that it is not possible with distance  $D$ ; that is,  $|f_D| < n$ . We copy the same reasoning from earlier and indeed, this implies that  $|f_{D'}| \leq |f_D|$  for all  $D' \leq D$ . However, this implies that  $|f_{D'}| < n$  and thus, it is also not possible for any distance smaller than  $D$ .

This shows that we have a monotonic behaviour. We now show that there are suitable lower and upper bounds. Since  $F$  is the farthest distance from any child, *every* child would be willing to travel with distance  $F$ . Therefore, it is unnecessary to check any distance greater than  $F$  and so, a suitable upper bound is  $F$ . A suitable lower bound is trivially 1. Thus, we have a suitable range to apply binary search, which proves that binary search is applicable.

We now prove the correctness of the algorithm. From the above discussion, the binary search procedure correctly traverses onto the appropriate subarray and so, finds the minimum value of  $D$ . The correctness of the subroutine comes directly from the correctness of **3.2**. Thus, the algorithm correctly returns the minimum value of  $D$  such that all  $n$  children on the nice list receives at least one present.

**Time complexity:** Since the binary search has a search space of size  $O(F)$ , our binary search performs  $\log F$  iterations. In each iteration, we perform the algorithm from **3.2** which has running time  $O(nm(k + n))$ . Therefore, the running time of the algorithm is  $O(nm(k + n) \log F)$ .

**Due Friday 28<sup>th</sup> of July at 6pm Sydney time (week 9)**

In this assignment we will apply dynamic programming. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of “one sympathy mark please” for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the [Ed forum](#). We will maintain a [FAQ thread](#) for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on [Overleaf](#), and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. ‘General’ here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is [UNSW Smarthinking](#), which is the university’s official writing support service.

Please review the UNSW policy on [plagiarism](#). Academic misconduct carries severe penalties.

Please read the [Frequently Asked Questions](#) document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

## Question 1

You are travelling with your friends along the Geraldton coast with cities  $c_0, c_1, c_2, \dots, c_n$  on the shore. You are starting in city  $c_0$  where a famous spa is, and need to reach the airport situated in city  $c_n$ , so you will visit each city  $c_0, c_1, c_2, \dots, c_n$  in that order.

In each city you may swap the animal you are riding on and the choices are a giraffe, a mammoth, an ant, an iguana, and a lemur, denoted  $G, M, A, I, L$  respectively. However, each city has its own rules what kind of animal exchanges are allowed. For example, in some of the cities you can swap a giraffe only for a lemur or an ant (and you **cannot** remain on your giraffe), in others you can swap a mammoth only for a giraffe or decide to remain on your mammoth, and so on. You know all the rules of all the cities  $c_1, \dots, c_n$ , expressed by a function  $R(i, a, b)$  where  $R(i, a, b) = 1$  if in city  $c_i$  one can swap animal  $a$  for animal  $b$ , and zero otherwise ( $a$  and  $b$  belong to the set  $\{G, M, A, I, L\}$ , and  $1 \leq i < n$ ). You also know the speed  $v(a)$  in km/h ( $a \in \{G, M, A, I, L\}$ ) of each of the five animals, as well as the distances  $d(i)$  in km between cities  $c_{i-1}$  and  $c_i$  for all  $i = 1, 2, \dots, n$ . Calculating a given value of  $R$ ,  $v$ , or  $d$  can be done in  $O(1)$  time.

You may begin your journey from  $c_0$  on any of the five animals. You need to choose which animals to use for each of the  $n$  trips between cities, such that your travel time is minimised and every animal swap is valid.

**1.1 [2 marks]** Consider the case of  $n = 2$ , with  $d(1) = 1$ ,  $d(2) = 3$ , and  $v$  defined as

$$v(G) = 7, \quad v(M) = 3, \quad v(A) = 2, \quad v(I) = 1, \quad v(L) = 4.$$

Define  $R$  so that

$$R(1, G, M) = 1, \quad R(1, A, M) = 1, \quad R(1, A, A) = 1, \quad R(1, I, G) = 1, \quad R(1, L, I) = 1,$$

and  $R(1, a, b) = 0$  for all other values of  $a, b$ .

Determine which animals you should choose on each trip in order to minimise the total time taken to travel from  $c_0$  to  $c_2$ . You *must* justify your selection.

Considering the compatibilities given by  $R$ , the only valid trips are:

|                      |                          |
|----------------------|--------------------------|
| Giraffe then Mammoth | $1/7 + 3/3 \approx 1.14$ |
| Ant then Mammoth     | $1/2 + 3/3 = 1.5$        |
| Ant then Ant         | $1/2 + 3/2 = 2$          |
| Iguana then Giraffe  | $1/1 + 3/7 \approx 1.43$ |
| Lemur then Iguana    | $1/4 + 3/1 = 3.25$       |

Of these, Giraffe from  $c_0$  to  $c_1$  and Mammoth from  $c_1$  to  $c_2$  is the fastest.

**1.2 [12 marks]** Design an algorithm which determines the minimal amount of time (in hours) it will take to get from  $c_0$  to  $c_n$  without making invalid swaps, as well as the animals required on each trip to achieve this time. Your algorithm must run in  $O(n)$  time.

[A] For each  $i$  and each animal  $a = G, M, A, I, L$  we solve the problem “ $P(i, a) = \text{the minimal amount of travel time needed to arrive to city } c_i \text{ riding the last leg of your journey (from } c_{i-1} \text{ to } c_i \text{) on animal } a$ ”.

[B] Let  $\text{opt}(i, a)$  denote the minimal amount of travel time needed to reach city  $c_i$  riding the last leg on animal  $a$ , and let  $A(i, a)$  be the animal which you swapped for animal  $a$  in city  $c_{i-1}$  during such fastest journey.

The base cases are given by  $\text{opt}(1, a) = d(1)/v(a)$  for  $a = G, M, A, I, L$ . For  $2 \leq i \leq n$  we

have recursion

$$\begin{aligned}\text{opt}(i, a) &= \min\{\text{opt}(i-1, x) + d(i)/v(a) : R(i-1, x, a) = 1; x = G, M, A, I, L\}; \\ A(i, a) &= \arg \min\{\text{opt}(i-1, x) + d(i)/v(a) : R(i-1, x, a) = 1; x = G, M, A, I, L\}.\end{aligned}$$

- [C] The final solution is given by  $\min\{\text{opt}(n, x) : x = G, M, A, I, L\}$ ; to obtain an optimal swapping strategy we let  $\alpha(n) = \arg \min\{\text{opt}(n, x) : x = G, M, A, I, L\}$ , and then backtrack, i.e., define recursively  $\alpha(n-i) = A(n-i, \alpha(n-(i-1)))$  for all  $1 \leq i \leq n$ . Note that  $\alpha(0)$  is the animal you should start your journey with in  $c_0$ .

- [D] Clearly, the procedure takes  $O(n)$  many steps.

**1.3 [6 marks]** While planning your trip, your friend Mae points out that animals are living creatures, and do, in fact, need to rest. To take this into consideration, you estimate the effect of consecutive trips on the animals, and come up with a magical constant,  $0 < \varepsilon < 1$ . For each consecutive trip an animal makes, the speed of the animal is multiplied by this constant.

For example, if you decide to travel on a mammoth from  $c_0$  to  $c_1$  to  $c_2$  to  $c_3$  without ever changing animals, then the mammoth's speed will be  $v(M)$  from  $c_0$  to  $c_1$ ,  $\varepsilon v(M)$  from  $c_1$  to  $c_2$ , and  $\varepsilon^2 v(M)$  from  $c_2$  to  $c_3$ . Of course, if you then swap animals, and later decide to travel by mammoth again, the new mammoths are not tired, and thus have speed  $v(M)$ . You may not “swap” an animal for “new” animals of the same type.

Design an algorithm which determines the minimal amount of time (in hours) it will take to get from  $c_0$  to  $c_n$  without making invalid swaps. Your friends are judging you for not considering this in the first place, so your algorithm must run in  $O(n^2)$  time before things become more awkward.

The fastest way to reach city  $i$  may involve taking a slower than optimal sequence of animals to city  $i-1$ . However, if we consider only the routes that end in a sequence of the same animal a certain number of times, we can optimally solve a more restricted subproblem.

- [A] For each  $i, k$  and each animal  $a = G, M, A, I, L$  we solve the problem “ $P(i, k, a) = \text{the minimal amount of travel time needed to arrive to city } c_i \text{ riding the last leg of your journey (from } c_{i-1} \text{ to } c_i\text{) on animal } a, \text{ and having ridden the same animal for } k \text{ previous legs (inclusive of this one)}$ ”.

- [B] Let  $\text{opt}(i, k, a)$  denote the minimal amount of travel time needed to reach city  $c_i$  riding the last leg on animal  $a$ , with the same animal ridden  $k$  times.

The base cases are given by  $\text{opt}(1, 1, a) = d(1)/v(a)$  for all  $a$ , and  $\text{opt}(1, k, a) = \infty$  for all  $a$  and  $1 < k \leq n$ . For  $2 \leq i \leq n$  and  $1 \leq k \leq i$  we have two cases for recursion. If  $k \geq 2$ ,

$$\text{opt}(i, k, a) = \begin{cases} \infty, & \text{if } R(i-1, a, a) = 0; \\ \text{opt}(i-1, k-1, a) + d(i)/\varepsilon^{k-1} v(a), & \text{otherwise.} \end{cases}$$

Otherwise, if  $k = 1$ ,

$$\text{opt}(i, 1, a) = \min_{x, k'} \{\text{opt}(i-1, k', x) + d(i)/v(a) : R(i-1, x, a) = 1, x \neq a, 1 \leq k' < i\}.$$

- [C] The final solution is given by  $\min_{k, a} \{\text{opt}(n, k, a)\}$ .

[D] For  $k > 1$  the subproblems take  $O(1)$  time to solve, and there are  $O(n^2)$  such subproblems. For  $k = 1$  the subproblems take  $O(n)$  time to solve, and there are  $O(n)$  such subproblems. So, overall, the time complexity is  $O(n^2)$

## Question 2

You are given a string  $w = w_1 w_2 \dots w_n$  of  $n$  letters that come from a fixed alphabet. A *palindrome* is a substring  $w'$  such that  $w'$  can be read the same forwards and backwards. For example,  $w' = kayak$  forms a palindrome while  $w'' = kayaak$  does not form a palindrome. A *palindromic substring* is a contiguous subsequence of  $w$  that forms a palindrome.

**2.1 [8 marks]** We want to eventually find a minimum cost decomposition of  $w$  into palindromes. However, to do this, we need to first find *where* the palindromes are. Given a string  $w = w_1 \dots w_n$ , describe an  $O(n^2)$  algorithm to identify all of the palindromic substrings of  $w$ .

**Hint.** How many subproblems can you have at most?

- There are faster algorithms such as a modification to *Manacher's algorithm* but if you use the algorithm, you will have to state the entire algorithm, and prove its correctness and running time.

Let  $\text{palindrome}(i, j)$  denote whether the substring  $w_i \dots w_j$  forms a palindrome. We note that  $w_i \dots w_j$  is a palindrome if and only if  $w_i = w_j$  and  $w_{i+1} \dots w_{j-1}$  forms a palindrome. Therefore, if  $w_i \neq w_j$ , then we can return `false`. Otherwise, we return the result of  $\text{palindrome}(i + 1, j - 1)$ . This can be compactly expressed as

$$\text{palindrome}(i, j) = (w_i = w_j) \wedge \text{palindrome}(i + 1, j - 1).$$

The base cases occur with  $\text{palindrome}(i, i) = \text{true}$  for each  $i = 1, \dots, n$  and  $\text{palindrome}(i, j) = \text{false}$  if  $j < i$ . We order the subproblems in increasing order of  $j - i$ ; that is, we solve the subproblems in increasing order of size of the substrings.

We, thus, have an  $O(n^2)$ -sized dynamic programming table where  $\text{palindrome}(i, j)$  determines whether the substring  $w_i \dots w_j$  forms a palindrome. We return all of the subproblems that return `true`.

To compute each subproblem, we just require an  $O(1)$  call to the dynamic programming table. Therefore, the dynamic programming solution takes  $O(n^2)$  to construct the table. We then read all of the  $O(n^2)$  cells, which takes  $O(n^2)$  in total.

**2.2 [12 marks]** A *palindromic decomposition* of a string  $w$  is a partition of  $w$  into substrings  $u_1, \dots, u_m$  such that  $w = u_1 \dots u_m$  and each part  $u_i$  forms a palindrome. For example, if  $w = \text{abcbabab}$ , then  $u_1 = \text{a}$ ,  $u_2 = \text{bcb}$ ,  $u_3 = \text{a}$ ,  $u_4 = \text{baab}$  is a palindromic decomposition of  $w$ .

We assign each substring  $u_i$  a cost denoted by  $\text{cost}(|u_i|)$ , where  $|u_i|$  is the length of  $u_i$ . Finally, we define the cost of a decomposition  $u_1, \dots, u_m$  to be the sum of the costs of the individual substrings  $u_i$ ; that is, if  $w = u_1 \dots u_m$ , then the cost of the decomposition is

$$\text{cost}(|u_1|) + \dots + \text{cost}(|u_m|).$$

We want to find the minimum cost of a *palindromic decomposition* of  $w$ . Given a string  $w = w_1 \dots w_n$  and an array of all costs, design an  $O(n^2)$  algorithm to compute the minimum cost of a palindromic decomposition of  $w$ .

**Hint.** Perform a pre-processing step using 2.1.

We begin by performing some preprocessing as before by finding all palindromes of  $w$  in  $O(n^2)$  time. Let  $\text{palindrome}(i, j)$  denote whether the substring  $w_i \dots w_j$  is a palindrome. Again, we define  $C(i, j)$  as follows:

$$C(i, j) = \begin{cases} \text{cost}(|w_i \dots w_j|) & \text{if } \text{palindrome}(i, j) = \text{true}, \\ \infty & \text{otherwise.} \end{cases}$$

Let  $\text{opt}(i)$  be the minimum cost of a palindrome decomposition of the substring  $w_1 \dots w_i$ . The recurrence is similar to Tutorial 6, Problem 3. Observe that, to split the string  $w_1 \dots w_i$  into palindromes, we require that some substring  $w_{j+1} \dots w_i$ , for some  $j < i$ , be a palindrome and the remaining string  $w_1 \dots w_j$  is the minimum cost decomposition to split  $w_1 \dots w_j$  into palindromes. This gives us

$$\text{opt}(i) = \min_{0 \leq j < i} \{\text{opt}(j) + C(j+1, i)\}.$$

This gives us the following decomposition.

$$\underbrace{w_1 \dots w_j}_{\text{opt}(j)} \quad \underbrace{w_{j+1} \dots w_i}_{\text{palindrome of size } j-i}$$

The base case is  $\text{opt}(0) = 0$  with the final solution being  $\text{opt}(n)$ . We solve the subproblems in increasing order of  $i$ . To analyse the time complexity, we observe that there are  $n$  subproblems. Each subproblem has at most  $O(n)$  work since we need to check at most  $n$  subproblems. Therefore, the running time is  $O(n^2)$ .

### Question 3

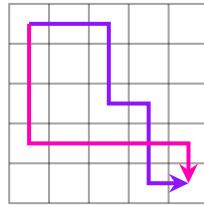
You are participating in a robotics competition, and need to program a robot to move through an  $m$  by  $n$  grid. The robot starts at cell  $(1, 1)$ , and must reach cell  $(m, n)$  to finish.

Your robot accepts two types of instructions:

- Start moving down at cell  $(i, j)$ ,
- Start moving right at cell  $(i, j)$ .

An instruction is *not* required to tell the robot which way to move at the start, as there is only one possibility based on the first turning instruction.

For example, in this grid with  $m = n = 5$ , the purple path requires four instructions (down at  $(1, 3)$ , right at  $(3, 3)$ , down at  $(3, 4)$ , right at  $(4, 4)$ ) while the pink path requires two (right at  $(4, 1)$ , down at  $(4, 5)$ ).



**3.1 [6 marks]** Design an algorithm that runs in  $O(mn)$  time and determines the number of distinct paths through the grid that the robot can traverse.

**Hint.** A closed form solution is possible, but we naturally encourage the use of Dynamic Programming which will assist with the rest of the question. A similar problem will also be discussed in the Week 8 Tutorial. Consider the 4 ways to reach a cell  $(i, j)$ .

## DP Solution

### Subproblems

Let  $P(i, j)$  be the number of distinct paths from  $(1, 1)$  to  $(i, j)$ .

### Recurrence

To reach cell  $(i, j)$  we can either

- Go to cell  $(i - 1, j)$ , then move down to cell  $(i, j)$ , or
- Go to cell  $(i, j - 1)$ , then move right to cell  $(i, j)$ .

Clearly, these cases are disjoint and cover all possible paths.

This gives our recurrence: for all  $1 < i \leq m$  and  $1 < j \leq n$ ,

$$P(i, j) = P(i - 1, j) + P(i, j - 1).$$

### Base cases

There is exactly one path to  $(1, 1)$ , so  $P(1, 1) = 1$ .

There is only one way to reach a cell in the top row (continuously moving right), so  $P(1, j) = 1$ . Similarly  $P(i, 1) = 1$ .

### Final answer

The answer is the number of paths to  $(m, n)$ , given by  $P(m, n)$ .

### Order of computation

We solve in increasing order of  $i$  then  $j$ , i.e. lexicographical order.

### Time complexity

We solve  $mn$  subproblems, each in  $O(1)$ , so the algorithm runs in  $O(mn)$ .

### Closed Form Solution

Any path through the grid must contain  $m - 1$  moves down and  $n - 1$  moves right. Each unique arrangement of these moves gives a unique path through the grid. The number of paths is then given by the permutations with repetitions formula:

$$\frac{(m+n-2)!}{(m-1)!(n-1)!}$$

Evaluating the factorials requires  $O(m+n) \subset O(mn)$  multiplications.

**3.2 [8 marks]** Unfortunately due to rising inflation your robot now only has a limited amount of memory, and can only store  $r$  instructions. Design an algorithm that runs in  $O(mnr)$  time and determines the number of distinct paths through the grid that the robot can traverse using at most  $r$  instructions.

There are four ways to reach a cell  $(i, j)$  when you consider the second last and third last cells in the path. Two of these ways use an instruction, while the other two don't.

### Subproblems

Let  $P(i, j, k, d)$  be the number of paths from  $(1, 1)$  to  $(i, j)$  which uses at most  $k$  instructions, and reach cell  $(i, j)$  by moving in direction  $d \in \{\rightarrow, \downarrow\}$ .

We can equivalently define two separate subproblems  $P(i, j, k)$  and  $Q(i, j, k)$ , where  $P(i, j, k)$  is the number of paths ending in a right move and  $Q(i, j, k)$  is the number ending in a down move.

### Recurrences

There are four disjoint ways we can reach  $(i, j)$ :

- First reach  $(i - 1, j)$  by moving down, then continue moving down to reach  $(i, j)$
- First reach  $(i, j - 1)$  by moving right, then continue moving right to reach  $(i, j)$
- First reach  $(i - 1, j)$  by moving right, then switch to moving down to reach  $(i, j)$
- First reach  $(i, j - 1)$  by moving down, then switch to moving right to reach  $(i, j)$ .

The first two options do not require an instruction as they continue in the previous direction, while the last two require instructions. To ensure that we are able to make the turn to reach  $(i, j)$ , we must ensure that at most  $k - 1$  instructions are used to reach the previous cell.

This gives our recurrences: For all  $1 < i \leq m$ ,  $1 < j \leq n$  and  $1 < k \leq r$ ,

$$\begin{aligned} P(i, j, k, \rightarrow) &= P(i, j - 1, k, \rightarrow) + P(i, j - 1, k - 1, \downarrow) \\ P(i, j, k, \downarrow) &= P(i - 1, j, k, \downarrow) + P(i - 1, j, k - 1, \rightarrow) \end{aligned}$$

### Base cases

There is only one path from  $(1, 1)$  to  $(1, 1)$  - the empty path - so  $P(1, 1, k, d) = 1$ .

The only way to reach a cell in row 1 is by continuously moving right, so  $P(1, j, k, \rightarrow) = 1$  and  $P(1, j, k, \downarrow) = 0$ . Similarly,  $P(i, 1, k, \downarrow) = 1$  and  $P(i, 1, k, \rightarrow) = 0$ .

Finally, it is impossible to reach any cell that is not in row or column 1 without any instructions, so  $P(i, j, 0, d) = 0$ .

### Final answer

The final answer is the number of ways to reach  $(m, n)$  using at most  $r$  instructions, reaching  $(m, n)$  from either direction. Therefore, the answer is given by  $P(m, n, r, \rightarrow) + P(m, n, r, \leftarrow)$ .

### Order of computation

Solve in increasing order of  $k$ ,  $i$  then  $j$ , solving for both directions for each  $(i, j, k)$ . That is,

$$\text{for } k = 1 \dots r \{ \text{for } i = 1 \dots m \{ \text{for } j = 1 \dots n \{ \text{solve } P(i, j, k, \rightarrow), P(i, j, k, \downarrow) \} \} \}.$$

### Time complexity

We solve  $2mnr$  subproblems, each in  $O(1)$  time, so the algorithm runs in  $O(mnr)$ .

**3.3 [6 marks]** To make the competition more interesting, Blake has placed obstacles in some of the cells, which the robot is unable to pass through. They have given you an array  $O[1 \dots m][1 \dots n]$ , where  $O[i][j]$  is TRUE if cell  $(i, j)$  has an obstacle and FALSE otherwise. Of course, Blake has made sure that a robot can travel from  $(1, 1)$  to  $(m, n)$  by moving only right and down.

You wish to find the smallest amount of memory  $r$  that your robot needs to be able to traverse the grid.

Design an algorithm that runs in  $O(mnr)$  time and finds the smallest number of instructions needed to traverse the grid.

Note that the parameter  $r$  in the time complexity is also the answer your algorithm is trying to find. This is similar to the Ford-Fulkerson algorithm for max flow, where the time complexity is dependent on the max flow for the given graph.

An algorithm that runs in  $O(mn(m + n))$  will be eligible for at most 4 marks for this part.

We adjust the answer to part 1, by adding the condition that  $P(i, j, k, d) = 0$  if  $O[i][j]$  is TRUE, as it is impossible to end on a cell containing an obstacle.

### Recurrences

For all  $1 < i \leq m$ ,  $1 < j \leq n$  and  $1 < k \leq r$ ,

$$P(i, j, k, \rightarrow) = \begin{cases} 0, & \text{if } O[i][j]; \\ P(i, j - 1, k, \rightarrow) + P(i, j - 1, k - 1, \downarrow), & \text{otherwise.} \end{cases}$$

$$P(i, j, k, \downarrow) = \begin{cases} 0, & \text{if } O[i][j]; \\ P(i - 1, j, k, \downarrow) + P(i - 1, j, k - 1, \rightarrow), & \text{otherwise.} \end{cases}$$

The base cases are unchanged.

### Computation and answer

We solve in increasing order of  $k, i$  then  $j$  as in part 1. However, we stop immediately at the first value of  $k$  for which  $P(m, n, k, \rightarrow) + P(m, n, k, \downarrow) \geq 1$ , because this is the first time we can reach the end, and hence must be the least number instructions possible to reach the end, so the value of  $k$  when this occurs must be  $r$ , in which is our answer.

### Time complexity

For each value of  $k$ , we solve  $2mn$  subproblems, each in constant time. We only run our algorithm until reaching the end is possible, which occurs when  $k$  is  $r$ , so there must be  $2mnr$  subproblems, each of which taking  $O(1)$  time. So the algorithm runs in  $O(mnr)$ .



# Algorithms:

## COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au  
office: 504 (CSE building K 17)

Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering  
University of New South Wales Sydney

Flow Networks Practice Problems

Several families are coming to a birthday celebration in a restaurant. You have arranged that  $v$  many tables will serve only vegetarian dishes,  $p$  many tables will serve dishes with meat but without pork and  $r$  many remaining tables will serve food with pork. You know that  $V$  many families are all vegetarians,  $P_1$  many families do not eat pork but do not mind eating vegetarian dishes,  $P_2$  many families do not eat pork but hate vegetarian dishes. Also  $R_1$  many families have no dietary restrictions and would also not mind eating vegetarian dishes or food without pork,  $R_2$  many families have no dietary restrictions but hate vegetarian dishes but can eat food without pork. Finally,  $S$  many families are from Serbia and cannot imagine not eating pork. You are also given the number of family members in each family and the number of seats at each table. In total, there are  $m$  families and  $n$  tables. You must place the guests at the tables so that their food preferences are respected and no two members from the same family sit at the same table. Your algorithm must run in time polynomial in  $m$  and  $n$ , and in case the problem has no solutions, your algorithm should output no solution.

We begin by constructing a bipartite flow network as follows:

- the left hand side  $n$  vertices represent  $n$  families,
- the right hand side  $m$  vertices represent  $m$  tables,
- a source  $s$  and a sink  $t$ ,
- connect  $s$  to each family vertex with an edge of capacity equal to the number of family members,
- connect each table vertex to  $t$  with an edge of capacity equal to the number of seats at that table,
- connect each family vertex with all tables compatible with the dietary preference with an edge of capacity 1

From this flow network construction, we run Edmonds-Karp to find the maximum flow. If the maximum flow is less than the total number of guests, then we output no solution. Otherwise, if the maximum flow equals the total number of guests, then a placement is possible.

Further, we can deduce a placement of guests at tables by examining which of the (family,table) edges carry flow: if there is 1 unit of flow from family  $i$  to table  $j$  then we seat a member of family  $i$  at table  $j$ .

The time complexity is  $O(|V||E|^2)$  where  $|V| = m + n + 2$  and  $|E| \leq m + n + mn$ , so algorithm runs in time polynomial in  $m$  and  $n$ .

You have been told of the wonder and beauty of a very famous painting. It is painted in the hyper-modern style, and so it is simply an  $n \times n$  grid of squares, with each square coloured either black or white. You have never seen this picture for yourself but have been told some details of it by a friend. Your friend has told you the value of  $n$  and the number of white squares in each row and each column. Additionally, your friend has also been kind enough to tell you the specific colour of some squares: some squares are black, some are white, and the rest they simply could not remember.

The more details they tell you, the more amazing this painting becomes but you begin to wonder that perhaps its simply too good to be true. Thus, you wish to design an algorithm which runs in time polynomial in  $n$  and determines whether or not such a painting can exist.

**Solution:** This problem can be viewed as a (bipartite) network flow problem in disguise. We begin by adjusting the count of the number of unknown white squares in each row and column based on the location of the known (white) squares. Note. The sum of the count in all rows must equal the sum of the count in all columns. Let this sum be  $S$  so we can infer that there are precisely  $S$  white squares among the squares of unknown colour. We then consider the bipartite graph where every row is a vertex on the left side of the graph and every column is a vertex on the right side, making  $2n$  vertices in total. Every square in the grid which is of unknown colour forms a directed edge from its corresponding row to its corresponding column. We can then convert this graph into a flow network as follows:

- each edge has capacity of 1.
- we add a source  $s$  and sink  $t$  to this graph.
- we add a directed edge from  $s$  to each row with capacity equal to the adjusted number of white squares in that row
- we add a directed edge from each column to  $t$  with capacity equal to the adjusted number of white squares in that column

It is evident that the saturated edges of the bipartite graph in any integer-valued flow from the source to the sink describe a possible colouring of the grid.

Therefore we can now find the maximum flow  $f$  via the Edmonds-Karp algorithm and as any such flow has a capacity at most  $S$ , a painting exists if and only if  $f = S$ .

With  $2n$  vertices in total,  $2n$  edges for  $s$  and  $t$  and  $n^2$  edges for the bipartite flow graph, we can conclude that our solution runs in a time complexity of  $O(n^5)$ , which is clearly polynomial in  $n$ .

Alice is the manager of a cafe which supplies  $n$  different kinds of drink and  $m$  different kinds of dessert. One day the materials are in short supply, so she can only make  $a_i$  cups of each drink type  $i$  and  $b_j$  servings of each dessert type  $j$ . On this day,  $k$  customers come to the cafe and the  $i^{th}$  of them has  $p_i$  favourite drinks ( $c_{i,1}, c_{i,2}, \dots, c_{i,p_i}$ ) and  $q_i$  favourite desserts ( $d_{i,1}, d_{i,2}, \dots, d_{i,q_i}$ ). Each customer wants to order one cup of any one of their favourite drinks and one serving of any one of their favourite desserts. If all their favourite drinks or all their favourite desserts are unavailable, the customer will instead leave the cafe and provide a poor rating. Alice wants to save the restaurants rating. From her extensive experience with these  $k$  customers, she has listed out the favourite drinks and desserts of each customer, and she wants your help to decide which customers orders should be fulfilled. Design an algorithm which runs in time polynomial in  $n, m$  and  $k$  and determines the smallest possible number of poor ratings that Alice can receive, given that:

- (a) all  $p_i$  and all  $q_i$  are 1 (i.e. each customer has only one favourite drink and one favourite dessert),
- (b) there is no restriction on the  $p_i$  and  $q_i$ .

**Solution:** (a) Construct a flow graph with a vertex  $A_i$  for each drink, a vertex  $B_j$  for each dessert, then two extra vertices for a source  $S$  and a sink  $T$ . For each drink  $i$ , add an edge with capacity  $a_i$  from  $S$  to  $A_i$ . For each dessert  $j$ , add an edge with capacity  $b_j$  from  $B_j$  to  $T$ . Finally, for each customer, add an edge of capacity 1 from  $c_{i,1}$  to  $d_{i,1}$ . The answer is  $k$  minus the maximum flow, found using Edmonds-Karp. The time complexity is  $O(|V||E|^2)$ , where  $|V| = 2 + n + m$  and  $|E| = n + m + k$ , so it is polynomial in  $n, m$  and  $k$ .

**Solution:** (b) We start by constructing a flow graph with:

- Two vertices,  $S$  and  $T$ , for the source and the sink.
- A vertex  $A_i$  for each drink  $i$ , with an edge of capacity  $a_i$  from  $S$  to  $A_i$ , to restrict the number of available cups of this drink.
- A vertex  $B_j$  for each dessert  $j$ , with an edge of capacity  $b_j$  from  $B_j$  to  $T$ , to restrict the number of available servings of this dessert.
- Two vertices  $C_i$  and  $D_i$  for each customer, with an edge of capacity 1 from  $C_i$  to  $D_i$  to ensure that each customer either has both their drink and dessert, or has neither. Note that we ignore serving them only one, as that is equivalent to serving them nothing in terms of ratings.
- For each favourite drink  $c_{i,j}$  of customer  $i$ , an edge of capacity 1 from  $A_{c_{i,j}}$  to  $C_i$  for any drink they would accept.
- For each favourite dessert  $d_{i,j}$  of customer  $i$ , an edge of capacity 1 from  $D_i$  to  $B_{d_{i,j}}$  for any dessert they would accept.

Each unit of flow through this graph assigns a different customer one of their favourite drinks and one of their favourite desserts.

Running the Edmonds-Karp algorithm on this graph then gives us the maximum flow, i.e. the maximum number of customers that we can satisfy, and so  $k$  minus this value is the minimum number of poor ratings. Our flow graph has  $|V| = 2 + n + m + 2k$  vertices and  $|E| \leq n + m + k + (n + m)k$  edges, so the time complexity of  $O(|V||E|^2)$  is indeed polynomial in  $n, m$  and  $k$ .

There are  $n$  cities (labelled  $1, 2, \dots, n$ ), connected by  $m$  bidirectional roads. Each road connects two different cities. A pair of cities may be connected by multiple roads. A well-known criminal is currently in city 1 and wishes to get to the city  $n$  via road. To catch them, the police have decided to block the minimum number of roads possible to make it impossible to get from city 1 to city  $n$ . However, some roads are major roads. In order to avoid disruption, the police cannot close any major roads. Your goal is to find the minimum number of roads to block to prevent the criminal from going from city 1 to city  $n$ , or report that the police cannot stop the criminal. Design an algorithm which achieves this goal and runs in time polynomial in  $n$  and  $m$ .

**Solution:** We construct a flow network as follows:

- create vertices  $v_1, v_2, \dots, v_n$  corresponding to  $n$  cities;
- make  $v_1$  as the source and  $v_n$  as the sink;
- suppose there are  $k$  roads between two cities  $i$  and  $j$ .
  - if none of the roads is a major road, we connect  $v_i$  and  $v_j$  with two directed edges in opposite directions and of capacity each equal to  $k$ .
  - if one of the roads is a major road the capacity of the two directed edges is set to  $m + 1$  ( $m$  is the total number of roads).
- We can now find the maximum flow  $f$  in such a network using the Edmonds-Karp algorithm, and recall that the value of this flow equals the capacity of the min cut.
- If  $|f| > m$  then at least one edge of capacity  $m + 1$  has been crossed, which indicates that every cut is crossed by a major road which cannot be blocked and thus we cannot catch the criminal.
- On the other hand, if  $|f| \leq m$ , then there is a cut which is crossed only by minor roads, so the criminal can be caught.
- To block the fewest number of roads, we block those roads which cross the min cut in the forward direction, i.e. those which go from a vertex reachable from  $v_1$  in the final residual graph to a vertex without this property.

Assume that you are given a network flow graph with  $n$  vertices, including a source  $s$ , a sink  $t$  and two other distinct vertices  $u$  and  $v$ , and  $m$  edges. Design an algorithm which runs in time polynomial in  $n$  and  $m$  and returns the smallest capacity-cut among all cuts for which:

- (a) the vertex  $u$  is on the same side of the cut as the source  $s$  and vertex  $v$  is on the same side as the sink  $t$ .
- (b) vertices  $u$  and  $v$  are at the same side of the cut.

**Solution:**

- (a) Add to the flow network one edge of infinite capacity from  $s$  to  $u$  and one edge of infinite capacity from  $v$  to  $t$ . Find the min cut in such a network. Run Edmonds-Karp to find the maximum flow and its corresponding minimal cut. Time complexity is  $O(nm^2)$ .
- (b) Given the flow network, we construct two directed edges, one from  $u$  to  $v$  and the other from  $v$  to  $u$ , both of which having infinite edge capacities respectively. We note that, if only one infinite edge is constructed from  $u$  to  $v$ , then  $v$  can still end up on the source side and  $u$  can still end up on the sink side, so the edge will not belong in the cut. From the discussion above, the two directed edges will ensure that  $u$  and  $v$  remain in the same side of the cut.

Assume that you are given a network flow graph with  $n$  vertices, including a source  $s$ , a sink  $t$  and two other distinct vertices  $u$  and  $v$ , and  $m$  edges. Design an algorithm which runs in time polynomial in  $n$  and  $m$  and returns the smallest capacity-cut among all cuts for which:

- (a) the vertex  $u$  is on the same side of the cut as the source  $s$  and vertex  $v$  is on the same side as the sink  $t$ .
- (b) vertices  $u$  and  $v$  are at the same side of the cut.

### Solution:

- (a) Add to the flow network one edge of infinite capacity from  $s$  to  $u$  and one edge of infinite capacity from  $v$  to  $t$ . Find the min cut in such a network. Run Edmonds-Karp to find the maximum flow and its corresponding minimal cut. Time complexity is  $O(nm^2)$ .
- (b) Given the flow network, we construct two directed edges, one from  $u$  to  $v$  and the other from  $v$  to  $u$ , both of which having infinite edge capacities respectively. We note that, if only one infinite edge is constructed from  $u$  to  $v$ , then  $v$  can still end up on the source side and  $u$  can still end up on the sink side, so the edge will not belong in the cut. From the discussion above, the two directed edges will ensure that  $u$  and  $v$  remain in the same side of the cut.

Assume that you are given a network flow graph with  $n$  vertices, including a source  $s$ , a sink  $t$  and two other distinct vertices  $u$  and  $v$ , and  $m$  edges. Design an algorithm which runs in time polynomial in  $n$  and  $m$  and returns the smallest capacity-cut among all cuts for which:

- (a) the vertex  $u$  is on the same side of the cut as the source  $s$  and vertex  $v$  is on the same side as the sink  $t$ .
- (b) vertices  $u$  and  $v$  are at the same side of the cut.

### Solution:

- (a) Add to the flow network one edge of infinite capacity from  $s$  to  $u$  and one edge of infinite capacity from  $v$  to  $t$ . Find the min cut in such a network. Run Edmonds-Karp to find the maximum flow and its corresponding minimal cut. Time complexity is  $O(nm^2)$ .
- (b) Given the flow network, we construct two directed edges, one from  $u$  to  $v$  and the other from  $v$  to  $u$ , both of which having infinite edge capacities respectively. We note that, if only one infinite edge is constructed from  $u$  to  $v$ , then  $v$  can still end up on the source side and  $u$  can still end up on the sink side, so the edge will not belong in the cut. From the discussion above, the two directed edges will ensure that  $u$  and  $v$  remain in the same side of the cut.

You know that  $n + 2$  spies  $S, s_1, s_2, \dots, s_n$  and  $T$  are communicating through  $m$  communication channels; in fact, for each  $i$  and each  $j$  you know if there is a channel through which spy  $s_i$  can send a secret message to spy  $s_j$  or if there is no such a channel (i.e., you know what the graph with spies as vertices and communication channels as edges looks like). Design an algorithm which runs in time polynomial in  $n$  and  $m$  that prevents spy  $S$  from sending a message to spy  $T$  by:

- (a) compromising as few channels as possible;
- (b) bribing as few of the other spies as possible.

### Solution:

- (a) We construct a flow network with each vertex  $v_i$  representing each spy  $i$  and with edges of capacity 1 from  $v_i$  to  $v_j$  if there is a communication link from  $i$  to  $j$ . We then run Edmonds-Karp on the flow network to find the maximum flow and the corresponding minimum cut. As any cut of the graph represents a valid set of channels (or edges) for our problem, then the edges that cross the minimum cut forwards are the ones that have to be compromised. This runs in polynomial time.

(b) We replicate the graph of part (a), but change all capacities of the edges to  $\infty$ . Then for each non-source and non-sink vertex  $v_i$ , we split  $v_i$  into two vertices  $v_i^{in}$  and  $v_i^{out}$  and connect  $v_i^{in}$  and  $v_i^{out}$  with an edge of capacity 1. For each edge incoming to  $v_i$ , we connect it to  $v_i^{in}$ ; for each edge outgoing from  $v_i$ , we modify it to instead be outgoing from  $v_i^{out}$ . Using this flow network graph, we repeat the same process as part (a) by running Edmonds-Karp and computing the minimum cut to find all the edges that represent the corresponding spies to bribe. As we have  $E = m + n$  and  $f \leq n$ , our total complexity is slightly different but still polynomial in  $m$  and  $n$  with  $O(E|f|) = n(m + n)$ .

You are manufacturing integrated circuits from a rectangular silicon board that is divided into an  $m \times n$  grid of squares. Each integrated circuit requires two adjacent squares, either vertically or horizontally, that are cut out from this board. However, some squares in the silicon board are defective and cannot be used for integrated circuits. For each pair of coordinates  $(i, j)$ , you are given a boolean  $d_{i,j}$  representing whether the square in row  $i$  and column  $j$  is defective or not. Design an algorithm which runs in time polynomial in  $m$  and  $n$  and determines the maximum number of integrated circuits that can be cut out from this board.

**Solution:** Form a bipartite graph with the left side consisting of all cells  $(x, y)$  which are non-defective and which satisfy that  $x + y$  is even and the right side consisting of all non-defective cells which satisfy that  $x + y$  is odd. Connect each cell with (at most four) adjacent non-defective cells. Now the problem reduces to finding a maximal matching in this graph which ensures that each cell is used in at most one IC.

You are manufacturing integrated circuits from a rectangular silicon board that is divided into an  $m \times n$  grid of squares. Each integrated circuit requires two adjacent squares, either vertically or horizontally, that are cut out from this board. However, some squares in the silicon board are defective and cannot be used for integrated circuits. For each pair of coordinates  $(i, j)$ , you are given a boolean  $d_{i,j}$  representing whether the square in row  $i$  and column  $j$  is defective or not. Design an algorithm which runs in time polynomial in  $m$  and  $n$  and determines the maximum number of integrated circuits that can be cut out from this board.

**Solution:** Form a bipartite graph with the left side consisting of all cells  $(x, y)$  which are non-defective and which satisfy that  $x + y$  is even and the right side consisting of all non-defective cells which satisfy that  $x + y$  is odd. Connect each cell with (at most four) adjacent non-defective cells. Now the problem reduces to finding a maximal matching in this graph which ensures that each cell is used in at most one IC.

You are given an  $n \times n$  chessboard with  $k$  white bishops on the board at the given cells  $(a_i, b_i)$ , where  $1 \leq a_i, b_i \leq n$  for each  $1 \leq i \leq k$ . You have to determine the largest number of black rooks which you can place on the board so that no two rooks are in the same row or in the same column or are under the attack of any of the  $k$  white bishops. Recall that bishops attack diagonally.

**Solution:** We construct a bipartite graph with  $n$  left vertices  $r_i$  representing  $n$  rows of the board and  $n$  right vertices  $c_j$  representing  $n$  columns of the board. We construct edges in such a graph so that vertex  $r_i$  is connected with a vertex  $r_j$  just in case the cell  $(i, j)$  on the board is not under attack of any of the bishops. We add a super source  $s$  and connect it with all vertices  $r_i$  with edges of capacity 1; we also add a super sink and connect all vertices  $c_j$  also with edges of capacity 1. The maximal number of rooks that meet the conditions is equal to the max flow in this flow network, with rooks placed in the cells corresponding to the occupied edges from  $r_i$  to  $c_j$ .

You are given an  $n \times n$  chessboard with  $k$  white bishops on the board at the given cells  $(a_i, b_i)$ , where  $1 \leq a_i, b_i \leq n$  for each  $1 \leq i \leq k$ . You have to determine the largest number of black rooks which you can place on the board so that no two rooks are in the same row or in the same column or are under the attack of any of the  $k$  white bishops. Recall that bishops attack diagonally.

**Solution:** We construct a bipartite graph with  $n$  left vertices  $r_i$  representing  $n$  rows of the board and  $n$  right vertices  $c_j$  representing  $n$  columns of the board. We construct edges in such a graph so that vertex  $r_i$  is connected with a vertex  $r_j$  just in case the cell  $(i, j)$  on the board is not under attack of any of the bishops. We add a super source  $s$  and connect it with all vertices  $r_i$  with edges of capacity 1; we also add a super sink and connect all vertices  $c_j$  also with edges of capacity 1. The maximal number of rooks that meet the conditions is equal to the max flow in this flow network, with rooks placed in the cells corresponding to the occupied edges from  $r_i$  to  $c_j$ .

You have  $n$  warehouses and  $n$  shops. At each warehouse, a truck is loaded with enough goods to supply one shop. There are  $m$  roads, each going from a warehouse to a shop, and driving along the  $i$ -th road takes  $d_i$  hours, where  $d_i$  is an integer. Design a polynomial time algorithm to send the trucks to the shops, minimising the time until all shops are supplied.

**Solution:** We combine a binary search with a maximum matching problem. We sort the values  $d_i$  in an increasing order. Represent each of the warehouse with a vertex and each shop also with a vertex. For any value of  $d_i$  we can connect all warehouses with all shops which are at most  $d_i$  away from each other, and then use a max-flow algorithm to determine if such a graph has a perfect matching of size  $n$ . Use binary search to find the smallest  $d_i$  for which such a perfect matching exists.

You have  $n$  warehouses and  $n$  shops. At each warehouse, a truck is loaded with enough goods to supply one shop. There are  $m$  roads, each going from a warehouse to a shop, and driving along the  $i$ -th road takes  $d_i$  hours, where  $d_i$  is an integer. Design a polynomial time algorithm to send the trucks to the shops, minimising the time until all shops are supplied.

**Solution:** We combine a binary search with a maximum matching problem. We sort the values  $d_i$  in an increasing order. Represent each of the warehouse with a vertex and each shop also with a vertex. For any value of  $d_i$  we can connect all warehouses with all shops which are at most  $d_i$  away from each other, and then use a max-flow algorithm to determine if such a graph has a perfect matching of size  $n$ . Use binary search to find the smallest  $d_i$  for which such a perfect matching exists.





# Algorithms:

## COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au

office: 504 (CSE building K 17)

Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering  
University of New South Wales Sydney

More Greedy Practice Problems

1. There are  $N$  robbers who have stolen  $N$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

**Solution:**

- Order the items so that their values are increasing.
- Take the lowest value item  $v_1$  and consider all thieves such that  $v_1$  is in their range of acceptable values.
- Pick the one with the smallest upper limit  $U_i$  and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value  $v_2$  and the remaining available thieves choosing for  $v_2$  the thief among them with the smallest upper bound  $U_j$  for which  $v_2 \in [L_j, U_j]$ , and so fourth.

1. There are  $N$  robbers who have stolen  $N$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

### Solution:

- Order the items so that their values are increasing.
- Take the lowest value item  $v_1$  and consider all thieves such that  $v_1$  is in their range of acceptable values.
- Pick the one with the smallest upper limit  $U_i$  and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value  $v_2$  and the remaining available thieves choosing for  $v_2$  the thief among them with the smallest upper bound  $U_j$  for which  $v_2 \in [L_j, U_j]$ , and so fourth.

1. There are  $N$  robbers who have stolen  $N$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

### Solution:

- Order the items so that their values are increasing.
- Take the lowest value item  $v_1$  and consider all thieves such that  $v_1$  is in their range of acceptable values.
- Pick the one with the smallest upper limit  $U_i$  and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value  $v_2$  and the remaining available thieves choosing for  $v_2$  the thief among them with the smallest upper bound  $U_j$  for which  $v_2 \in [L_j, U_j]$ , and so fourth.

1. There are  $N$  robbers who have stolen  $N$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

### Solution:

- Order the items so that their values are increasing.
- Take the lowest value item  $v_1$  and consider all thieves such that  $v_1$  is in their range of acceptable values.
- Pick the one with the smallest upper limit  $U_i$  and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value  $v_2$  and the remaining available thieves choosing for  $v_2$  the thief among them with the smallest upper bound  $U_j$  for which  $v_2 \in [L_j, U_j]$ , and so fourth.

1. There are  $N$  robbers who have stolen  $N$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that either distributes the items so that each robber is happy or determines that there is no such distribution.

### Solution:

- Order the items so that their values are increasing.
- Take the lowest value item  $v_1$  and consider all thieves such that  $v_1$  is in their range of acceptable values.
- Pick the one with the smallest upper limit  $U_i$  and give the first item to that thief.
- Continue in this manner considering the item with the next smallest value  $v_2$  and the remaining available thieves choosing for  $v_2$  the thief among them with the smallest upper bound  $U_j$  for which  $v_2 \in [L_j, U_j]$ , and so fourth.

We now need to prove that this method is optimal.

- For each thief  $i$  let  $L_i$  be their lowest acceptable value and let  $U_i$  be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy.
- This means there is at least one item assignment which violates our greedy assignment policy. Let item  $k$  with value  $v_k$  be the least valuable such item, and suppose that this item was assigned to thief  $i$  (so  $v_k \in [L_i, U_i]$ ).
- Since this item assignment violated our greedy policy, there must be another thief  $j$  (with range  $[L_j, U_j]$ ) who would have been happy with item  $k$ , but whose highest acceptable value is lower than thief  $i$ 's, so  $v_k \in [L_j, U_j]$  and  $U_j < U_i$ .

We now need to prove that this method is optimal.

- For each thief  $i$  let  $L_i$  be their lowest acceptable value and let  $U_i$  be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy.
- This means there is at least one item assignment which violates our greedy assignment policy. Let item  $k$  with value  $v_k$  be the least valuable such item, and suppose that this item was assigned to thief  $i$  (so  $v_k \in [L_i, U_i]$ ).
- Since this item assignment violated our greedy policy, there must be another thief  $j$  (with range  $[L_j, U_j]$ ) who would have been happy with item  $k$ , but whose highest acceptable value is lower than thief  $i$ 's, so  $v_k \in [L_j, U_j]$  and  $U_j < U_i$ .

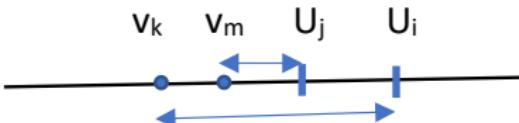
We now need to prove that this method is optimal.

- For each thief  $i$  let  $L_i$  be their lowest acceptable value and let  $U_i$  be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy.
- This means there is at least one item assignment which violates our greedy assignment policy. Let item  $k$  with value  $v_k$  be the least valuable such item, and suppose that this item was assigned to thief  $i$  (so  $v_k \in [L_i, U_i]$ ).
- Since this item assignment violated our greedy policy, there must be another thief  $j$  (with range  $[L_j, U_j]$ ) who would have been happy with item  $k$ , but whose highest acceptable value is lower than thief  $i$ 's, so  $v_k \in [L_j, U_j]$  and  $U_j < U_i$ .

We now need to prove that this method is optimal.

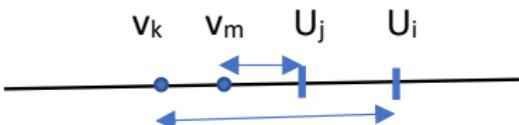
- For each thief  $i$  let  $L_i$  be their lowest acceptable value and let  $U_i$  be their highest acceptable value. Assume that there is an assignment of items to thieves which satisfies all thieves, but which is different to that obtained by our greedy strategy.
- This means there is at least one item assignment which violates our greedy assignment policy. Let item  $k$  with value  $v_k$  be the least valuable such item, and suppose that this item was assigned to thief  $i$  (so  $v_k \in [L_i, U_i]$ ).
- Since this item assignment violated our greedy policy, there must be another thief  $j$  (with range  $[L_j, U_j]$ ) who would have been happy with item  $k$ , but whose highest acceptable value is lower than thief  $i$ 's, so  $v_k \in [L_j, U_j]$  and  $U_j < U_i$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.



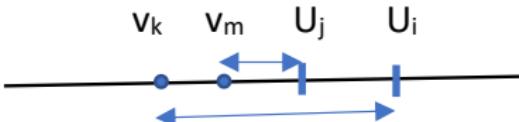
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.



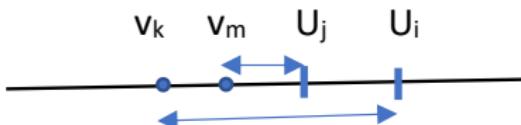
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all thieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.



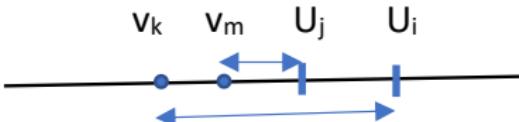
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.



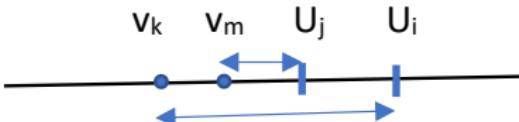
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all tieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.



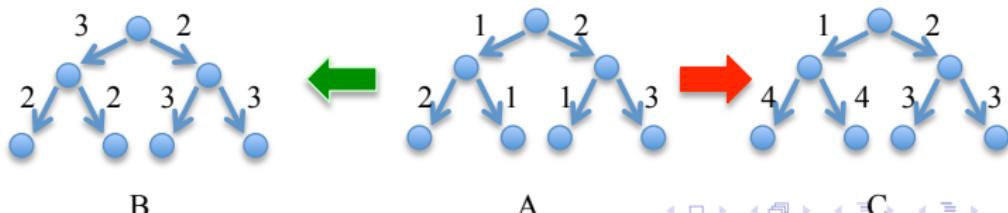
- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all thieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

- Now suppose this thief was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ .
- Hence,  $v_m \in [L_i, U_i]$ , which means that thief  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two thieves while keeping them both happy.

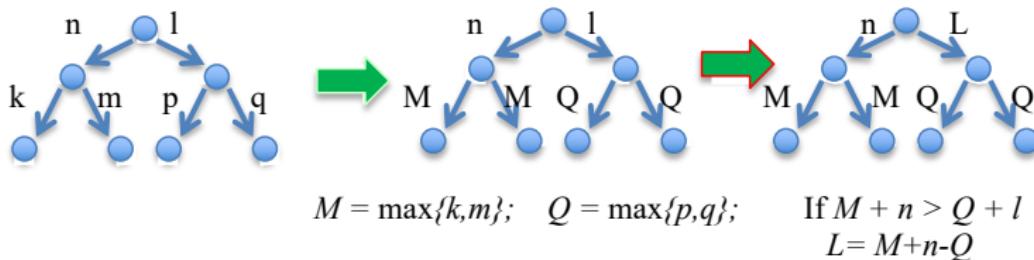


- By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy.
- Therefore, our method is optimal.
- Sorting items according to their value is done in  $O(n \log n)$ ; for each item we search through the list to thieves to find all thieves for whom that item is in their range which is done in  $O(n)$  for each item. Thus in total this takes  $O(n^2)$ .
- Thus the whole algorithm runs in time  $O(n^2)$ .

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

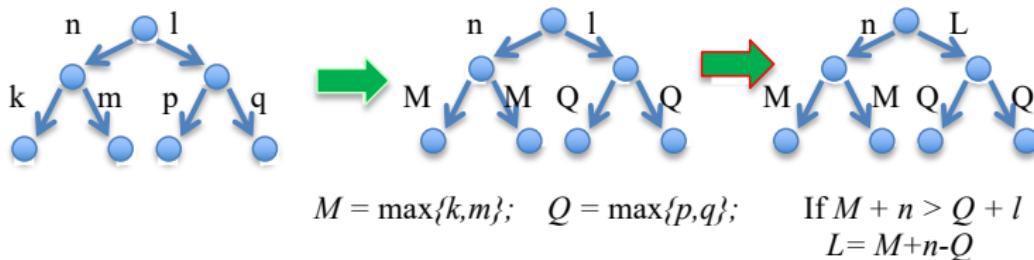


**Solution:** We proceed by recursion, working from the leaves towards the root. Consult the figure below.



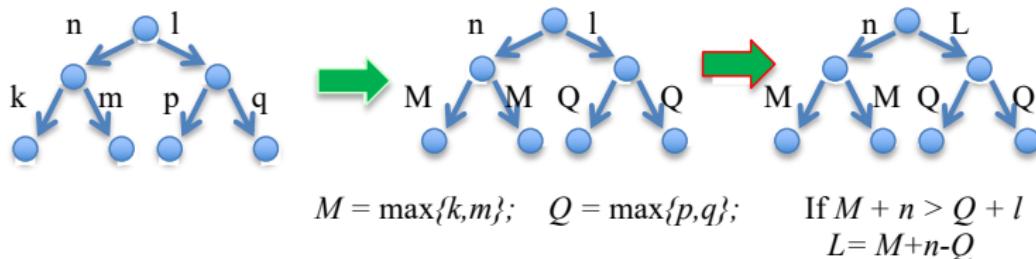
- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let  $M = \max\{k, m\}$  and  $Q = \max\{p, q\}$  and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider  $M + n$  and  $Q + l$ . If  $M + n > Q + l$ , then we increase  $l$  to  $L = M + n - Q$ , so that  $M + n = Q + L$ . If  $Q + l > M + n$ , then we increase  $n$  to  $N = Q + L - M$  so that  $M + N = Q + l$ .

**Solution:** We proceed by recursion, working from the leaves towards the root. Consult the figure below.

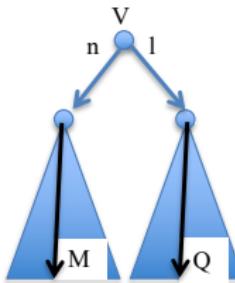


- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let  $M = \max\{k, m\}$  and  $Q = \max\{p, q\}$  and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider  $M + n$  and  $Q + l$ . If  $M + n > Q + l$ , then we increase  $l$  to  $L = M + n - Q$ , so that  $M + n = Q + L$ . If  $Q + l > M + n$ , then we increase  $n$  to  $N = Q + L - M$  so that  $M + N = Q + l$ .

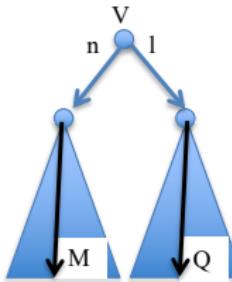
**Solution:** We proceed by recursion, working from the leaves towards the root. Consult the figure below.



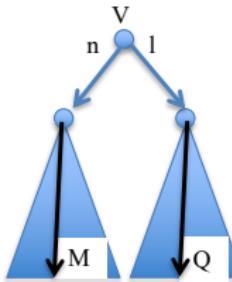
- We start with the edges connecting two adjacent leaves. Clearly, they have to be of the same length.
- Thus, referring to the figure, we let  $M = \max\{k, m\}$  and  $Q = \max\{p, q\}$  and change the shorter length of the two to the value of the longer length.
- Moving one level up, we consider  $M + n$  and  $Q + l$ . If  $M + n > Q + l$ , then we increase  $l$  to  $L = M + n - Q$ , so that  $M + n = Q + L$ . If  $Q + l > M + n$ , then we increase  $n$  to  $N = Q + L - M$  so that  $M + N = Q + l$ .



- We continue in this manner; assuming that we have made the lengths of all paths to all leaves of the subtree  $T_1$  equal to  $M$  and the lengths of all paths to all leaves of the subtree  $T_2$  equal to  $Q$ , we consider the edges connecting a vertex  $V$  to the roots of these subtrees.
- Assuming that the edge connecting  $V$  to the root of  $T_1$  is of length  $n$  and that the edge connecting  $V$  to the root of  $T_2$  is of length  $l$ ; if  $M + n > Q + l$  we replace the edge of length  $l$  with an edge of length  $M + n - Q$ ; if  $Q + l > M + n$  we replace the edge of length  $n$  with an edge of length  $Q + l - M$ .
- Clearly, the produced tree is of minimal total length with equal distances from the root to all leaves.



- We continue in this manner; assuming that we have made the lengths of all paths to all leaves of the subtree  $T_1$  equal to  $M$  and the lengths of all paths to all leaves of the subtree  $T_2$  equal to  $Q$ , we consider the edges connecting a vertex  $V$  to the roots of these subtrees.
- Assuming that the edge connecting  $V$  to the root of  $T_1$  is of length  $n$  and that the edge connecting  $V$  to the root of  $T_2$  is of length  $l$ ; if  $M + n > Q + l$  we replace the edge of length  $l$  with an edge of length  $M + n - Q$ ; if  $Q + l > M + n$  we replace the edge of length  $n$  with an edge of length  $Q + l - M$ .
- Clearly, the produced tree is of minimal total length with equal distances from the root to all leaves.



- We continue in this manner; assuming that we have made the lengths of all paths to all leaves of the subtree  $T_1$  equal to  $M$  and the lengths of all paths to all leaves of the subtree  $T_2$  equal to  $Q$ , we consider the edges connecting a vertex  $V$  to the roots of these subtrees.
- Assuming that the edge connecting  $V$  to the root of  $T_1$  is of length  $n$  and that the edge connecting  $V$  to the root of  $T_2$  is of length  $l$ ; if  $M + n > Q + l$  we replace the edge of length  $l$  with an edge of length  $M + n - Q$ ; if  $Q + l > M + n$  we replace the edge of length  $n$  with an edge of length  $Q + l - M$ .
- Clearly, the produced tree is of minimal total length with equal distances from the root to all leaves.

3. 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 denominations 4c, 3c and 1c and an amount to be paid of 6 cents. The greedy algorithm would first give one 4c coin and would then be forced to give 2 cents using two 1c coins. However, giving two 3c coins is more optimal.

3. 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 denominations 4c, 3c and 1c and an amount to be paid of 6 cents. The greedy algorithm would first give one 4c coin and would then be forced to give 2 cents using two 1c coins. However, giving two 3c coins is more optimal.

4. Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a subsequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ .

**Solution:**

- Use a simple greedy strategy.
- First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ .
- Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter.
- If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a subsequence of  $A$ .

4. Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a subsequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ .

### Solution:

- Use a simple greedy strategy.
- First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ .
- Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter.
- If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a subsequence of  $A$ .

4. Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a subsequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ .

### Solution:

- Use a simple greedy strategy.
- First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ .
- Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter.
- If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a subsequence of  $A$ .

4. Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a subsequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ .

### Solution:

- Use a simple greedy strategy.
- First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ .
- Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter.
- If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a subsequence of  $A$ .

4. Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a subsequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ .

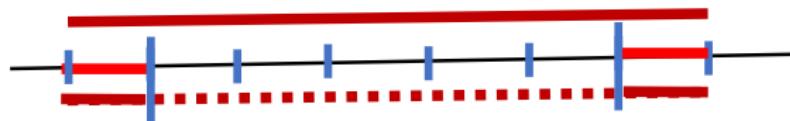
### Solution:

- Use a simple greedy strategy.
- First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ .
- Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter.
- If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a subsequence of  $A$ .

5. There is a line of 111 stalls, some of which lack a roof and 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:**

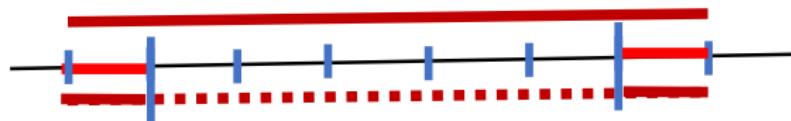
- First, cover all roofless stalls with a single board that stretches from the first roofless stall to the last roofless stall.
- Then, repeat the following up to 10 times: Find the longest line of consecutive roofed stalls that are covered by a board, and then excise that part of the board, replacing it with two boards.



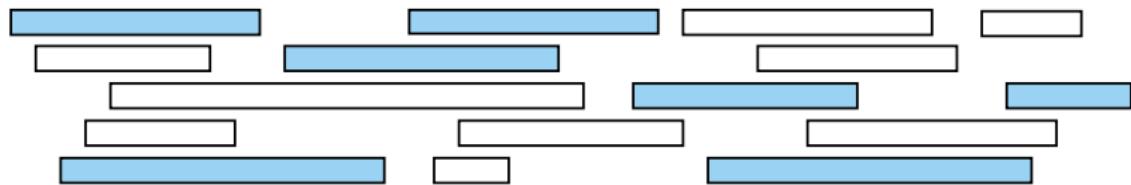
5. There is a line of 111 stalls, some of which lack a roof and need to be covered with boards. You can use up to 11 boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

### Solution:

- First, cover all roofless stalls with a single board that stretches from the first roofless stall to the last roofless stall.
- Then, repeat the following up to 10 times: Find the longest line of consecutive roofed stalls that are covered by a board, and then excise that part of the board, replacing it with two boards.



6. Let  $X$  be a set of  $n$  intervals on the real line. A subset of intervals  $Y \subseteq X$  is called a tiling path if the intervals in  $Y$  cover the intervals in  $X$ , that is, any real value that is contained in some interval in  $X$  is also contained in some interval in  $Y$ . The size of a tiling cover is just the number of intervals. Describe and analyse an algorithm to compute the smallest tiling path of  $X$  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  $X$ .



The seven shaded intervals form a tiling path.

## Solution:

- Sort the intervals in increasing order of their left endpoints.
- Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint.
- Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint.
- Continue in this manner until an interval with the absolute largest right endpoint is chosen.
- The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in  $O(n \log n)$  time.

## Solution:

- Sort the intervals in increasing order of their left endpoints.
- Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint.
- Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint.
- Continue in this manner until an interval with the absolute largest right endpoint is chosen.
- The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in  $O(n \log n)$  time.

## Solution:

- Sort the intervals in increasing order of their left endpoints.
- Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint.
- Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint.
- Continue in this manner until an interval with the absolute largest right endpoint is chosen.
- The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in  $O(n \log n)$  time.

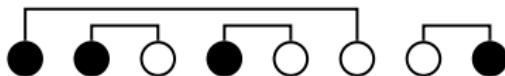
## Solution:

- Sort the intervals in increasing order of their left endpoints.
- Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint.
- Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint.
- Continue in this manner until an interval with the absolute largest right endpoint is chosen.
- The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in  $O(n \log n)$  time.

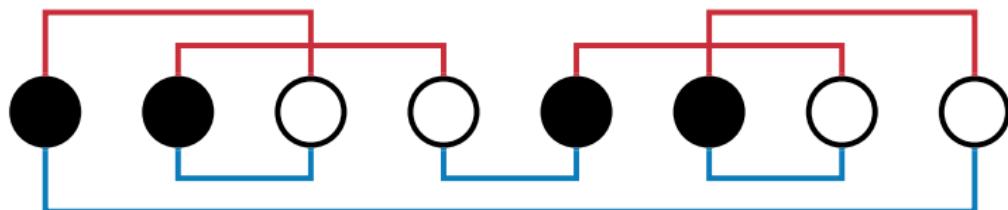
## Solution:

- Sort the intervals in increasing order of their left endpoints.
- Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint.
- Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint.
- Continue in this manner until an interval with the absolute largest right endpoint is chosen.
- The algorithm involves sorting the intervals, and then performing by a single pass through all of the intervals. Hence, the algorithm runs in  $O(n \log n)$  time.

7. Assume that you are given  $n$  white and  $n$  black dots lying in a random configuration on a straight line, equally spaced. Design a greedy algorithm which connects each black dot with a (different) white dot, so that the total length of wires used to form such connected pairs is minimal. The length of wire used to connect two dots is equal to the straight-line distance between them.

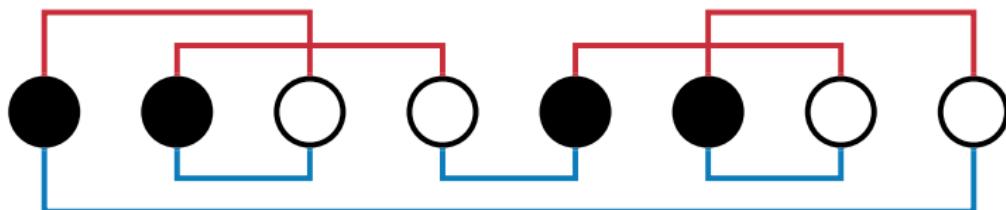


**Solution:** One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



- Connecting the closest pairs (blue lines) uses  $3 + 7 = 10$  units of length while the connections in red use only  $4 \times 2 = 8$  units of length.
- The correct approach is to go from left to right and connect the leftmost dot with the leftmost dot of the opposite colour and then continue in this way.

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



- Connecting the closest pairs (blue lines) uses  $3 + 7 = 10$  units of length while the connections in red use only  $4 \times 2 = 8$  units of length.
- The correct approach is to go from left to right and connect the leftmost dot with the leftmost dot of the opposite colour and then continue in this way.

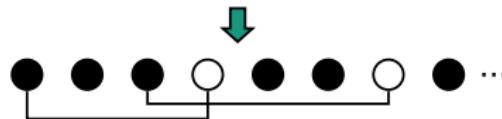
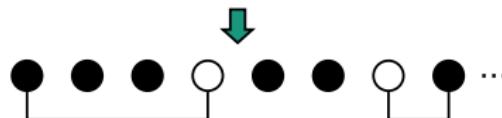
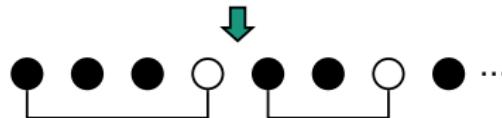
- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour.
- We look at the leftmost dot for which the greedy strategy is violated.
- There are three types of configurations to consider ( shown on the next slide) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy.
- Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.

- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour.
- We look at the leftmost dot for which the greedy strategy is violated.
- There are three types of configurations to consider ( shown on the next slide) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy.
- Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.

- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour.
- We look at the leftmost dot for which the greedy strategy is violated.
- There are three types of configurations to consider ( shown on the next slide) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy.
- Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.

- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour.
- We look at the leftmost dot for which the greedy strategy is violated.
- There are three types of configurations to consider ( shown on the next slide) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy.
- Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.

- To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots.
- Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour.
- We look at the leftmost dot for which the greedy strategy is violated.
- There are three types of configurations to consider ( shown on the next slide) - but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy.
- Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.



8. You are running a small manufacturing shop with plenty of workers, but just a single milling machine. You have to produce  $n$  items; item  $i$  requires  $m_i$  machining time first, then  $p_i$  polishing time by hand; finally it is packaged by hand and this takes  $c_i$  amount of time. The machine can mill only one item at a time, but your workers can polish and package 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:** Ignore the machining time and sort the jobs in decreasing order with respect to the sum  $p_i + c_i$ . The proof of optimality is straightforward. It is enough to show that if  $p_i + c_i < p_{i+1} + c_{i+1}$  and yet job  $i$  has been scheduled before job  $i + 1$ , then the completion time cannot increase if we swap jobs  $i$  and  $i + 1$ .

8. You are running a small manufacturing shop with plenty of workers, but just a single milling machine. You have to produce  $n$  items; item  $i$  requires  $m_i$  machining time first, then  $p_i$  polishing time by hand; finally it is packaged by hand and this takes  $c_i$  amount of time. The machine can mill only one item at a time, but your workers can polish and package 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:** Ignore the machining time and sort the jobs in decreasing order with respect to the sum  $p_i + c_i$ . The proof of optimality is straightforward. It is enough to show that if  $p_i + c_i < p_{i+1} + c_{i+1}$  and yet job  $i$  has been scheduled before job  $i + 1$ , then the completion time cannot increase if we swap jobs  $i$  and  $i + 1$ .

9. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if a job is accepted, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the activity selection problem.) Given a list of  $n$  such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in  $n$ . You may assume for simplicity that no two jobs have the same start or end times.

**Solution:** This problem is similar to the Activity Selection problem, except that time is now a 24hr circle, rather than an interval, because there might be jobs whose start time is before the midnight and finishing time after midnight. However, we can reduce it to several instances of the interval case.

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight.
- We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time.
- We record the number of accepted jobs obtained in this manner as  $a_0$ .
- We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ .
- We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ .
- Finally we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

- To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them.
- If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem.
- If it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .
- Time complexity: Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time.
- Each of the  $k + 1$  procedures run in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity).
- The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

10. Assume that you got a fabulous job and you wish to repay your student loan as quickly as possible. Unfortunately, the bank *Western Road Robbery* which gave you the loan has the condition that you must start your monthly repayments by paying off \$1 and then each subsequent month you must pay either double the amount you paid the previous month, the same amount as the previous month or half the amount you paid the previous month. On top of these conditions, your 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 minimises the number of months it will take you to repay your loan while satisfying all of the bank's requirements. If the optimality of your solution is obvious from the algorithm description, you do not have to provide any further correctness proof.

**Solution:** To repay your loan as quickly as possible, you will want to double your repayment as much as possible while still ensuring that you can finish your repayment with a payment of \$1 at the end.

10. Assume that you got a fabulous job and you wish to repay your student loan as quickly as possible. Unfortunately, the bank *Western Road Robbery* which gave you the loan has the condition that you must start your monthly repayments by paying off \$1 and then each subsequent month you must pay either double the amount you paid the previous month, the same amount as the previous month or half the amount you paid the previous month. On top of these conditions, your 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 minimises the number of months it will take you to repay your loan while satisfying all of the bank's requirements. If the optimality of your solution is obvious from the algorithm description, you do not have to provide any further correctness proof.

**Solution:** To repay your loan as quickly as possible, you will want to double your repayment as much as possible while still ensuring that you can finish your repayment with a payment of \$1 at the end.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

- So assuming that the value of your loan is  $S$ ; we first find the largest  $n$  such that  $2(1 + 2 + 2^2 + \dots + 2^n) \leq S$ .
- This amounts to finding the largest  $n$  such that  $2(2^{n+1} - 1) \leq S$ .
- Let  $R = S - 2(2^{n+1} - 1)$ . Since  $2(2^{n+2} - 1) > S$ , we get

$$R = S - 2(2^{n+1} - 1) < 2(2^{n+2} - 1) - 2(2^{n+1} - 1) = 2^{n+3} - 2^{n+2} = 2^{n+2}$$

i.e.,

$$R \leq 2^{n+2} - 1 = 1 + 2 + 2^2 + \dots + 2^{n+1}$$

- You can now return your loan as follows: start with \$1 and keep doubling until you reach  $2^n$ .
- If  $R \geq 2^{n+1}$  double once again; if this is the case let  $P = R - 2^{n+1}$ .
- Otherwise let  $P = R$ . Represent  $P$  in binary; you now start halving the amount you pay, but you repeat once the amount  $2^k$  whenever the  $k^{th}$  bit of  $P$  is one.

11. Alice wants to throw a party and is deciding whom to invite. She has  $n$  people to choose from, and she has created a list consisting of pairs of people who know each other. She wants to invite 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. Argue that your algorithm indeed produces a subset with the largest possible number of invitees.

## Solution:

- For each person, count the number of people they know and the number of people they do not know, recording this in a table.
- Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered.
- Then, update the count of known/unknown people for each affected person.
- Continue in this manner until everyone left satisfies the condition.
- This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

## Solution:

- For each person, count the number of people they know and the number of people they do not know, recording this in a table.
- Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered.
- Then, update the count of known/unknown people for each affected person.
- Continue in this manner until everyone left satisfies the condition.
- This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

## Solution:

- For each person, count the number of people they know and the number of people they do not know, recording this in a table.
- Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered.
- Then, update the count of known/unknown people for each affected person.
- Continue in this manner until everyone left satisfies the condition.
- This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

## Solution:

- For each person, count the number of people they know and the number of people they do not know, recording this in a table.
- Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered.
- Then, update the count of known/unknown people for each affected person.
- Continue in this manner until everyone left satisfies the condition.
- This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

## Solution:

- For each person, count the number of people they know and the number of people they do not know, recording this in a table.
- Now eliminate all people who know fewer than five other people or don't know fewer than five people among the set of people currently being considered.
- Then, update the count of known/unknown people for each affected person.
- Continue in this manner until everyone left satisfies the condition.
- This strategy is clearly optimal, as we begin with the set of all people, and only eliminate people when they do not satisfy the condition.

12. 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 direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll 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:**

- This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it.
- To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices.
- The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex.
- Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm.

12. 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 direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll 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:

- This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it.
- To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices.
- The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex.
- Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm.

12. 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 direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll 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:

- This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it.
- To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices.
- The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex.
- Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm.

12. 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 direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll 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:

- This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it.
- To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices.
- The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex.
- Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm.

12. 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 direct road, you have to spend a night in a hotel in all intermediate cities. You are given a map of Elbonia with toll 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:

- This is almost the shortest path problem, except that now going through a vertex (i.e., city) also has a weight associated with it.
- To reduce this problem to the shortest path problem, split each vertex (representing a city) into two new vertices.
- The first of these vertices will represent arrivals at that city, and will receive all the incoming edges of the original vertex, while the other will represent departures from that city, and will be the starting point of all outgoing edges from the original vertex.
- Connect the two new vertices with an edge whose weight is the price of spending the night in that city. Now use Dijkstra's algorithm.

13. 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 the total number of arcs, not in terms of the total length of these arcs). Prove that your solution is optimal.

**Solution:** This problem is equivalent to problem 9 about the processor. Convert each arc into a time period (by, for example, superimposing a 24-hour clock onto the unit circle) and proceed as in problem 9.

13. 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 the total number of arcs, not in terms of the total length of these arcs). Prove that your solution is optimal.

**Solution:** This problem is equivalent to problem 9 about the processor. Convert each arc into a time period (by, for example, superimposing a 24-hour clock onto the unit circle) and proceed as in problem 9.

14. 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 a 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:**

- To avoid confusion, we will imagine that the arcs are laid out on a straight line (with some arcs potentially wrapping around the ends).
- Let  $A_1$  be the arc whose right endpoint occurs earliest on the line,  $A_2$  be the arc whose right endpoint occurs next-earliest, and so on.
- Stab  $A_1$  at its right endpoint. Then, consider the arcs that have not yet been stabbed, and pick the arc whose right endpoint occurs the earliest (with respect to the last stabbed point) and stab it at its right endpoint.

14. 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 a 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:

- To avoid confusion, we will imagine that the arcs are laid out on a straight line (with some arcs potentially wrapping around the ends).
- Let  $A_1$  be the arc whose right endpoint occurs earliest on the line,  $A_2$  be the arc whose right endpoint occurs next-earliest, and so on.
- Stab  $A_1$  at its right endpoint. Then, consider the arcs that have not yet been stabbed, and pick the arc whose right endpoint occurs the earliest (with respect to the last stabbed point) and stab it at its right endpoint.

14. 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 a 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:

- To avoid confusion, we will imagine that the arcs are laid out on a straight line (with some arcs potentially wrapping around the ends).
- Let  $A_1$  be the arc whose right endpoint occurs earliest on the line,  $A_2$  be the arc whose right endpoint occurs next-earliest, and so on.
- Stab  $A_1$  at its right endpoint. Then, consider the arcs that have not yet been stabbed, and pick the arc whose right endpoint occurs the earliest (with respect to the last stabbed point) and stab it at its right endpoint.

14. 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 a 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:

- To avoid confusion, we will imagine that the arcs are laid out on a straight line (with some arcs potentially wrapping around the ends).
- Let  $A_1$  be the arc whose right endpoint occurs earliest on the line,  $A_2$  be the arc whose right endpoint occurs next-earliest, and so on.
- Stab  $A_1$  at its right endpoint. Then, consider the arcs that have not yet been stabbed, and pick the arc whose right endpoint occurs the earliest (with respect to the last stabbed point) and stab it at its right endpoint.

- Repeat in this manner until all arcs have been stabbed, and record the number of needles used as  $n_1$ .
- Now start over, but this time begin by stabbing  $A_2$  at its right endpoint, and then continue in the above manner, wrapping around the end of the line if necessary.
- Record the number of needles used as  $n_2$ . Do this for each arc, and then take the arrangement for which the number of needles used is minimal.

- Repeat in this manner until all arcs have been stabbed, and record the number of needles used as  $n_1$ .
- Now start over, but this time begin by stabbing  $A_2$  at its right endpoint, and then continue in the above manner, wrapping around the end of the line if necessary.
- Record the number of needles used as  $n_2$ . Do this for each arc, and then take the arrangement for which the number of needles used is minimal.

- Repeat in this manner until all arcs have been stabbed, and record the number of needles used as  $n_1$ .
- Now start over, but this time begin by stabbing  $A_2$  at its right endpoint, and then continue in the above manner, wrapping around the end of the line if necessary.
- Record the number of needles used as  $n_2$ . Do this for each arc, and then take the arrangement for which the number of needles used is minimal.

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

**Solution:** This problem is essentially identical to the problem of placing minimal numbers of base station towers which we did in class.

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

**Solution:** This problem is essentially identical to the problem of placing minimal numbers of base station towers which we did in class.

16. There are  $N$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $K$  people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

Solution:

- Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on.
- Note that the final group consisting of the fastest people may contain fewer than  $K$  people.
- Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).
- The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group.

16. There are  $N$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $K$  people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

### Solution:

- Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on.
- Note that the final group consisting of the fastest people may contain fewer than  $K$  people.
- Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).
- The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group.

16. There are  $N$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $K$  people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

### Solution:

- Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on.
- Note that the final group consisting of the fastest people may contain fewer than  $K$  people.
- Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).
- The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group.

16. There are  $N$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $K$  people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

### Solution:

- Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on.
- Note that the final group consisting of the fastest people may contain fewer than  $K$  people.
- Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).
- The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group.

16. There are  $N$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $K$  people, but you can only move as fast as the slowest person in each group. Design an algorithm to determine the fewest number of days you will need to guide everyone through the pass.

### Solution:

- Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on.
- Note that the final group consisting of the fastest people may contain fewer than  $K$  people.
- Then guide these groups one by one through the pass (the order in which you guide the groups does not matter).
- The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group.

17. There are  $N$  courses that you can take. Each course has a minimum required IQ, and will raise your IQ by a certain amount when you take it. Design an algorithm to find the minimum number of courses you will need to take to get an IQ of at least  $K$ .

**Solution:** Out of all the courses remaining that you can take, take the course that will raise your IQ the most. Repeat until your IQ is  $K$  or higher.

The optimality of this strategy should be obvious. Taking the course that will raise your IQ the most will make the most courses available to you, thus giving you more options.

17. There are  $N$  courses that you can take. Each course has a minimum required IQ, and will raise your IQ by a certain amount when you take it. Design an algorithm to find the minimum number of courses you will need to take to get an IQ of at least  $K$ .

**Solution:** Out of all the courses remaining that you can take, take the course that will raise your IQ the most. Repeat until your IQ is  $K$  or higher.

The optimality of this strategy should be obvious. Taking the course that will raise your IQ the most will make the most courses available to you, thus giving you more options.

17. There are  $N$  courses that you can take. Each course has a minimum required IQ, and will raise your IQ by a certain amount when you take it. Design an algorithm to find the minimum number of courses you will need to take to get an IQ of at least  $K$ .

**Solution:** Out of all the courses remaining that you can take, take the course that will raise your IQ the most. Repeat until your IQ is  $K$  or higher.

The optimality of this strategy should be obvious. Taking the course that will raise your IQ the most will make the most courses available to you, thus giving you more options.



# Algorithms:

## COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au

office: 504 (CSE building K 17)

Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering  
University of New South Wales Sydney

More Dynamic Programming Practice Problems

1. You are traveling by a canoe down a river and there are  $n$  trading posts along the way. Before starting your journey, you are given for each  $1 \leq i < j \leq n$  the fee  $F(i, j)$  for renting a canoe from post  $i$  to post  $j$ . These fees are arbitrary. For example it is possible that  $F(1, 3) = 10$  and  $F(1, 4) = 5$ . You begin at trading post 1 and must end at trading post  $n$  (using rented canoes). Your goal is to design an efficient algorithms which produces the sequence of trading posts where you change your canoe which minimizes the total rental cost.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

## Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post  $i$ .*
- The base case is  $\text{opt}(1) = 0$ .
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \quad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg \min_{1 \leq j < i} \{\text{opt}(j) + F(j, i)\}, \quad i > 1.$$

(Note:  $\arg \min$  returns the value of  $j$  that produces the minimal value of  $\text{opt}(j) + F(j, i)$ ).

- The minimum cost is  $\text{opt}(n)$ .
- To get the sequence, we backtrack from post  $n$  giving the sequence  $\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$ . Reverse this sequence.
- The complexity is  $O(n^2)$  because there are  $n$  subproblems, and each subproblem takes  $O(n)$  to find the best previous trading post.

2. You are given a set of  $n$  types of rectangular boxes, where the  $i^{th}$  box has height  $h_i$ , width  $w_i$  and depth  $d_i$ . You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

## Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives  $6n$  boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box  $B_1$  can go on top of a box  $B_0$  then  $B_0$  precedes box  $B_1$  in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number  $i$ ?*

## Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives  $6n$  boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box  $B_1$  can go on top of a box  $B_0$  then  $B_0$  precedes box  $B_1$  in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number  $i$ ?*

## Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives  $6n$  boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box  $B_1$  can go on top of a box  $B_0$  then  $B_0$  precedes box  $B_1$  in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number  $i$ ?*

## Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives  $6n$  boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box  $B_1$  can go on top of a box  $B_0$  then  $B_0$  precedes box  $B_1$  in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number  $i$ ?*

## Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives  $6n$  boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box  $B_1$  can go on top of a box  $B_0$  then  $B_0$  precedes box  $B_1$  in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number  $i$ ?*

- The recursion is:

$$\text{opt}(i) = \max\{\text{opt}(j)+h_i : \text{over all } j \text{ such that } w_j > w_i \text{ and } d_j > d_i\}.$$

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e.,  $\max_{1 \leq i \leq 6n} \text{opt}(i)$ .
- The complexity is  $O(n^2)$ . There are  $6n$  different subproblems, and each subproblem requires us to search through  $O(n)$  boxes to find ones that have a base large enough to stack the current box on top.

- The recursion is:

$$\text{opt}(i) = \max\{\text{opt}(j)+h_i : \text{over all } j \text{ such that } w_j > w_i \text{ and } d_j > d_i\}.$$

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e.,  $\max_{1 \leq i \leq 6n} \text{opt}(i)$ .
- The complexity is  $O(n^2)$ . There are  $6n$  different subproblems, and each subproblem requires us to search through  $O(n)$  boxes to find ones that have a base large enough to stack the current box on top.

- The recursion is:

$$\text{opt}(i) = \max\{\text{opt}(j)+h_i : \text{over all } j \text{ such that } w_j > w_i \text{ and } d_j > d_i\}.$$

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e.,  $\max_{1 \leq i \leq 6n} \text{opt}(i)$ .
- The complexity is  $O(n^2)$ . There are  $6n$  different subproblems, and each subproblem requires us to search through  $O(n)$  boxes to find ones that have a base large enough to stack the current box on top.

3. You have an amount of money  $M$  and you are in a candy store. There are  $n$  kinds of candies and for each candy you know how much pleasure you get by eating it, which is a number between 1 and 100, as well as the price of each candy. Your task is to choose which candies you are going to buy to maximise the total pleasure you will get by gobbling them all.

**Solution:** This is a knapsack problem with duplicated values. The pleasure score is the value of the item, the cost of a particular type of candy is its weight, and the money  $M$  is the capacity of the knapsack. The complexity is  $O(Mn)$ .

3. You have an amount of money  $M$  and you are in a candy store. There are  $n$  kinds of candies and for each candy you know how much pleasure you get by eating it, which is a number between 1 and 100, as well as the price of each candy. Your task is to choose which candies you are going to buy to maximise the total pleasure you will get by gobbling them all.

**Solution:** This is a knapsack problem with duplicated values. The pleasure score is the value of the item, the cost of a particular type of candy is its weight, and the money  $M$  is the capacity of the knapsack. The complexity is  $O(Mn)$ .

4. Consider a 2-D map with a horizontal river passing through its centre. There are  $n$  cities on the southern bank with  $x$ -coordinates  $a_1 \dots a_n$  and  $n$  cities on the northern bank with  $x$ -coordinates  $b_1 \dots b_n$ . You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. When connecting cities, you are only allowed to connect the  $i^{th}$  city on the northern bank to the  $i^{th}$  city on the southern bank.

**Solution:** Sort the cities on the south bank according to their  $x$  coordinates and then re-enumerate them so that they appear as  $\{1, 2, 3, \dots\}$ , and then apply the same permutation to the cities on the north bank. Now the problem reduces to finding a maximal increasing sequence of indices of cities on the north bank.

4. Consider a 2-D map with a horizontal river passing through its centre. There are  $n$  cities on the southern bank with  $x$ -coordinates  $a_1 \dots a_n$  and  $n$  cities on the northern bank with  $x$ -coordinates  $b_1 \dots b_n$ . You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. When connecting cities, you are only allowed to connect the  $i^{th}$  city on the northern bank to the  $i^{th}$  city on the southern bank.

**Solution:** Sort the cities on the south bank according to their  $x$  coordinates and then re-enumerate them so that they appear as  $\{1, 2, 3, \dots\}$ , and then apply the same permutation to the cities on the north bank. Now the problem reduces to finding a maximal increasing sequence of indices of cities on the north bank.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true and false xor true* such that it evaluates to true.

**Solution:**

- Let there be  $n$  symbols, and  $n - 1$  operations between them.
- We solve the following two subproblems:

*How many ways (denoted  $T(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to true ( $T$ )?*

and

*How many ways (denoted  $F(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to false ( $F$ )?*

- Problems are solved in the order of  $r - l$  breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true and false xor true* such that it evaluates to true.

### Solution:

- Let there be  $n$  symbols, and  $n - 1$  operations between them.
- We solve the following two subproblems:

*How many ways (denoted  $T(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to true ( $T$ )?*

and

*How many ways (denoted  $F(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to false ( $F$ )?*

- Problems are solved in the order of  $r - l$  breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true and false xor true* such that it evaluates to true.

### Solution:

- Let there be  $n$  symbols, and  $n - 1$  operations between them.
- We solve the following two subproblems:

*How many ways (denoted  $T(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to true ( $T$ )?*

and

*How many ways (denoted  $F(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to false ( $F$ )?*

- Problems are solved in the order of  $r - l$  breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true and false xor true* such that it evaluates to true.

### Solution:

- Let there be  $n$  symbols, and  $n - 1$  operations between them.
- We solve the following two subproblems:

*How many ways (denoted  $T(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to true ( $T$ )?*

and

*How many ways (denoted  $F(l, r)$ ) are there to place brackets to make the expression starting from at the  $l^{\text{th}}$  symbol and ending at  $r^{\text{th}}$  symbol evaluate to false ( $F$ )?*

- Problems are solved in the order of  $r - l$  breaking evens arbitrarily.

- The base case is that  $T(i, i)$  is 1 if symbol  $i$  is *true*, and 0 if symbol  $i$  is *false*. The reverse applies to  $F(i, i)$ .
- Otherwise, for each subproblem, we ‘split’ the expression around an operator  $m$  so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$T(l, r) = \sum_{m=l}^{r-1} T\text{Split}(l, m, r)$$

$$F(l, r) = \sum_{m=l}^{r-1} F\text{Split}(l, m, r)$$

- The base case is that  $T(i, i)$  is 1 if symbol  $i$  is *true*, and 0 if symbol  $i$  is *false*. The reverse applies to  $F(i, i)$ .
- Otherwise, for each subproblem, we ‘split’ the expression around an operator  $m$  so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$T(l, r) = \sum_{m=l}^{r-1} T\text{Split}(l, m, r)$$

$$F(l, r) = \sum_{m=l}^{r-1} F\text{Split}(l, m, r)$$

- The base case is that  $T(i, i)$  is 1 if symbol  $i$  is *true*, and 0 if symbol  $i$  is *false*. The reverse applies to  $F(i, i)$ .
- Otherwise, for each subproblem, we ‘split’ the expression around an operator  $m$  so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$T(l, r) = \sum_{m=l}^{r-1} T\text{Split}(l, m, r)$$

$$F(l, r) = \sum_{m=l}^{r-1} F\text{Split}(l, m, r)$$

- The base case is that  $T(i, i)$  is 1 if symbol  $i$  is *true*, and 0 if symbol  $i$  is *false*. The reverse applies to  $F(i, i)$ .
- Otherwise, for each subproblem, we ‘split’ the expression around an operator  $m$  so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$T(l, r) = \sum_{m=l}^{r-1} T\text{Split}(l, m, r)$$

$$F(l, r) = \sum_{m=l}^{r-1} F\text{Split}(l, m, r)$$

- The base case is that  $T(i, i)$  is 1 if symbol  $i$  is *true*, and 0 if symbol  $i$  is *false*. The reverse applies to  $F(i, i)$ .
- Otherwise, for each subproblem, we ‘split’ the expression around an operator  $m$  so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$T(l, r) = \sum_{m=l}^{r-1} T\text{Split}(l, m, r)$$

$$F(l, r) = \sum_{m=l}^{r-1} F\text{Split}(l, m, r)$$

$$\text{TSplit}(l, m, r) = \begin{cases} T(l, m) \times T(m + 1, r) & \text{if operator } m \text{ is 'and'}, \\ T(l, m) \times F(m + 1, r) + T(l, m) \times T(m + 1, r) + \\ F(l, m) \times T(m + 1, r) & \text{if operator } m \text{ is 'or'}, \\ F(l, m) \times F(m + 1, r) + F(l, m) \times T(m + 1, r) \\ & \text{if operator } m \text{ is 'xor'}. \end{cases}$$

$$\text{FSplit}(l, m, r) = \begin{cases} T(l, m) \times F(m + 1, r) + F(l, m) \times F(m + 1, r) + \\ F(l, m) \times T(m + 1, r) & \text{if operator } m \text{ is 'and'}, \\ F(l, m) \times F(m + 1, r) & \text{if operator } m \text{ is 'or'}, \\ T(l, m) \times T(m + 1, r) + F(l, m) \times F(m + 1, r) \\ & \text{if operator } m \text{ is 'xor'}. \end{cases}$$

The complexity is  $O(n^3)$ . There are  $O(n^2)$  different ranges that  $l$  and  $r$  could cover, and each needs the evaluations of TSplit or FSplit at up to  $n - 1$  different splitting points.

6. A company is organising a party for its employees. The organisers of the party want it to be a fun party, and so have assigned a fun rating to every employee. The employees are organised into a strict hierarchy, i.e., a tree rooted at the president. There is one restriction, though, on the guest list to the party: an employee and their immediate supervisor (parent in the tree) cannot both attend the party (because that would be no fun at all). Give an algorithm that makes a guest list for the party that maximises the sum of the fun ratings of the guests.

### Solution:

- Let us denote by  $T(i)$  the subtree of the tree  $T$  of all employees which is rooted at an employee  $i$ .
- For each such subtree we will compute two quantities,  $I(i)$  and  $E(i)$ .  $I(i)$  is the maximal sum of fun factors  $\text{fun}(i)$  of employees selected from subtree  $T(i)$  which satisfies the constraint and which must include the root  $i$ .
- $E(i)$  is the maximal sum of fun factors of employees selected from the subtree  $T(i)$  but which does NOT include  $i$ .

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee  $i$  whose (immediate) subordinates are  $j_1, \dots, j_m$  we have

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k)$$

$$E(i) = \sum_{1 \leq k \leq m} \max(E(j_k), I(j_k))$$

- Notice that in the definition of  $E(i)$  we have the option to either include the children of  $i$  or exclude them, whatever produces larger value of  $E(i)$ .
- The final answer is  $\max(I(n), E(n))$  where  $n$  is the root of the corporate tree  $T$ .
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee  $i$  whose (immediate) subordinates are  $j_1, \dots, j_m$  we have

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k)$$

$$E(i) = \sum_{1 \leq k \leq m} \max(E(j_k), I(j_k))$$

- Notice that in the definition of  $E(i)$  we have the option to either include the children of  $i$  or exclude them, whatever produces larger value of  $E(i)$ .
- The final answer is  $\max(I(n), E(n))$  where  $n$  is the root of the corporate tree  $T$ .
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee  $i$  whose (immediate) subordinates are  $j_1, \dots, j_m$  we have

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k)$$

$$E(i) = \sum_{1 \leq k \leq m} \max(E(j_k), I(j_k))$$

- Notice that in the definition of  $E(i)$  we have the option to either include the children of  $i$  or exclude them, whatever produces larger value of  $E(i)$ .
- The final answer is  $\max(I(n), E(n))$  where  $n$  is the root of the corporate tree  $T$ .
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee  $i$  whose (immediate) subordinates are  $j_1, \dots, j_m$  we have

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k)$$

$$E(i) = \sum_{1 \leq k \leq m} \max(E(j_k), I(j_k))$$

- Notice that in the definition of  $E(i)$  we have the option to either include the children of  $i$  or exclude them, whatever produces larger value of  $E(i)$ .
- The final answer is  $\max(I(n), E(n))$  where  $n$  is the root of the corporate tree  $T$ .
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee  $i$  whose (immediate) subordinates are  $j_1, \dots, j_m$  we have

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k)$$

$$E(i) = \sum_{1 \leq k \leq m} \max(E(j_k), I(j_k))$$

- Notice that in the definition of  $E(i)$  we have the option to either include the children of  $i$  or exclude them, whatever produces larger value of  $E(i)$ .
- The final answer is  $\max(I(n), E(n))$  where  $n$  is the root of the corporate tree  $T$ .
- Clearly such algorithm runs in time linear in the number of all employees.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

7. You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C$ , using as few bins as possible.

### Solution:

- We will solve subproblems  $P(i, j)$  of packing  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$  for all  $1 \leq i \leq n_1$  and all  $1 \leq j \leq n_2$ .
- Let  $C/s_1 = K$ . The recursion step is essentially an exhaustive search:

$$\text{opt}(i, j) = 1 + \min_{0 \leq k \leq K} \text{opt}(i - k, j - \lfloor (C - k s_1)/s_2 \rfloor)$$

- Thus, we try all options of placing between 0 and  $K$  many items of size  $s_1$  into one single box and then fill the box to capacity with items of size  $s_2$ , and optimally packing the remaining items.
- The algorithm runs in time  $O(K n_1 n_2)$ .
- Thus, such an algorithm runs in exponential time in the size of the input, because input takes only  $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$  many bits to represent.

8. You are given  $n$  activities and for each activity  $i$  you are given its starting time  $s_i$ , its finishing time  $f_i$  and the profit  $p_i$  which you get if you schedule this activity. Only one activity can take place at any time. Your task is to design an algorithm which produces a subset  $S$  of those  $n$  activities so that no two activities in  $S$  overlap and such that the sum of profits of all activities in  $S$  is maximised.

## Solution:

- Sort all activities by finishing time  $f_i$ .
- We solve the following subproblems for all  $i$ : *What is the maximum profit  $\text{opt}(i)$  we can make if we are to choose between activities  $a_1, a_2, \dots, a_i$  and such that activity  $a_i$  is the last activity we do?*
- Recursion is simple:

$$\begin{aligned}\text{opt}(i) &= \max\{\text{opt}(j) : f_j < s_i\} + p_i \\ \text{prev}(i) &= \arg \max\{\text{opt}(j) : f_j < s_i\}\end{aligned}$$

- Finally, the solution to the original problem is profit of  $\max_{1 \leq i \leq n} \text{opt}(i)$  and the sequence of jobs can be obtained starting with  $m = \arg \max_{1 \leq i \leq n} \text{opt}(i)$  and then backtracking via  $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \dots$

## Solution:

- Sort all activities by finishing time  $f_i$ .
- We solve the following subproblems for all  $i$ : *What is the maximum profit  $\text{opt}(i)$  we can make if we are to choose between activities  $a_1, a_2, \dots, a_i$  and such that activity  $a_i$  is the last activity we do?*
- Recursion is simple:

$$\begin{aligned}\text{opt}(i) &= \max\{\text{opt}(j) : f_j < s_i\} + p_i \\ \text{prev}(i) &= \arg \max\{\text{opt}(j) : f_j < s_i\}\end{aligned}$$

- Finally, the solution to the original problem is profit of  $\max_{1 \leq i \leq n} \text{opt}(i)$  and the sequence of jobs can be obtained starting with  $m = \arg \max_{1 \leq i \leq n} \text{opt}(i)$  and then backtracking via  $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \dots$

## Solution:

- Sort all activities by finishing time  $f_i$ .
- We solve the following subproblems for all  $i$ : *What is the maximum profit  $\text{opt}(i)$  we can make if we are to choose between activities  $a_1, a_2, \dots, a_i$  and such that activity  $a_i$  is the last activity we do?*
- Recursion is simple:

$$\begin{aligned}\text{opt}(i) &= \max\{\text{opt}(j) : f_j < s_i\} + p_i \\ \text{prev}(i) &= \arg \max\{\text{opt}(j) : f_j < s_i\}\end{aligned}$$

- Finally, the solution to the original problem is profit of  $\max_{1 \leq i \leq n} \text{opt}(i)$  and the sequence of jobs can be obtained starting with  $m = \arg \max_{1 \leq i \leq n} \text{opt}(i)$  and then backtracking via  $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \dots$

## Solution:

- Sort all activities by finishing time  $f_i$ .
- We solve the following subproblems for all  $i$ : *What is the maximum profit  $\text{opt}(i)$  we can make if we are to choose between activities  $a_1, a_2, \dots, a_i$  and such that activity  $a_i$  is the last activity we do?*
- Recursion is simple:

$$\begin{aligned}\text{opt}(i) &= \max\{\text{opt}(j) : f_j < s_i\} + p_i \\ \text{prev}(i) &= \arg \max\{\text{opt}(j) : f_j < s_i\}\end{aligned}$$

- Finally, the solution to the original problem is profit of  $\max_{1 \leq i \leq n} \text{opt}(i)$  and the sequence of jobs can be obtained starting with  $m = \arg \max_{1 \leq i \leq n} \text{opt}(i)$  and then backtracking via  $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \dots$

9. Your shipping company has just received  $N$  individual shipping requests (jobs). For each request  $i$ , you know it will require  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $T$  trucks in total. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

**Solution:**

- This is just the standard knapsack problem with  $t_i$  being the size of the  $i^{th}$  item,  $d_i$  its value and with  $T$  as the capacity of the knapsack.
- Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed.
- The complexity is  $O(NT)$ .

9. Your shipping company has just received  $N$  individual shipping requests (jobs). For each request  $i$ , you know it will require  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $T$  trucks in total. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is just the standard knapsack problem with  $t_i$  being the size of the  $i^{th}$  item,  $d_i$  its value and with  $T$  as the capacity of the knapsack.
- Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed.
- The complexity is  $O(NT)$ .

9. Your shipping company has just received  $N$  individual shipping requests (jobs). For each request  $i$ , you know it will require  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $T$  trucks in total. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is just the standard knapsack problem with  $t_i$  being the size of the  $i^{th}$  item,  $d_i$  its value and with  $T$  as the capacity of the knapsack.
- Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed.
- The complexity is  $O(NT)$ .

9. Your shipping company has just received  $N$  individual shipping requests (jobs). For each request  $i$ , you know it will require  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $T$  trucks in total. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is just the standard knapsack problem with  $t_i$  being the size of the  $i^{th}$  item,  $d_i$  its value and with  $T$  as the capacity of the knapsack.
- Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed.
- The complexity is  $O(NT)$ .

10. Again your shipping company has just received  $N$  individual shipping requests (jobs). This time, for each request  $i$ , you know it will require  $e_i$  employees and  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $E$  employees and  $T$  trucks in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

Solution:

- This is a slight modification of the knapsack problem with two constraints on the total size of all jobs.
- Think of a knapsack which can hold items of total weight not exceeding  $E$  units of weight and total volume not exceeding  $T$  units of volume, with item  $i$  having a weight of  $e_i$  integer units of weight and  $t_i$  integer units of volume.
- For each triplet  $(e, t, i)$  such that  $e \leq E$ ,  $t \leq T$ ,  $i \leq N$  we solve the following subproblem: *Choose a sub-collection of items  $1 \dots i$  that fits in a knapsack of capacity  $e$  units of weight and  $t$  units of volume, which is of largest possible value.*

10. Again your shipping company has just received  $N$  individual shipping requests (jobs). This time, for each request  $i$ , you know it will require  $e_i$  employees and  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $E$  employees and  $T$  trucks in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is a slight modification of the knapsack problem with two constraints on the total size of all jobs.
- Think of a knapsack which can hold items of total weight not exceeding  $E$  units of weight and total volume not exceeding  $T$  units of volume, with item  $i$  having a weight of  $e_i$  integer units of weight and  $t_i$  integer units of volume.
- For each triplet  $(e, t, i)$  such that  $e \leq E$ ,  $t \leq T$ ,  $i \leq N$  we solve the following subproblem: *Choose a sub-collection of items  $1 \dots i$  that fits in a knapsack of capacity  $e$  units of weight and  $t$  units of volume, which is of largest possible value.*

10. Again your shipping company has just received  $N$  individual shipping requests (jobs). This time, for each request  $i$ , you know it will require  $e_i$  employees and  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $E$  employees and  $T$  trucks in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is a slight modification of the knapsack problem with two constraints on the total size of all jobs.
- Think of a knapsack which can hold items of total weight not exceeding  $E$  units of weight and total volume not exceeding  $T$  units of volume, with item  $i$  having a weight of  $e_i$  integer units of weight and  $t_i$  integer units of volume.
- For each triplet  $(e, t, i)$  such that  $e \leq E$ ,  $t \leq T$ ,  $i \leq N$  we solve the following subproblem: *Choose a sub-collection of items  $1 \dots i$  that fits in a knapsack of capacity  $e$  units of weight and  $t$  units of volume, which is of largest possible value.*

10. Again your shipping company has just received  $N$  individual shipping requests (jobs). This time, for each request  $i$ , you know it will require  $e_i$  employees and  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $E$  employees and  $T$  trucks in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

### Solution:

- This is a slight modification of the knapsack problem with two constraints on the total size of all jobs.
- Think of a knapsack which can hold items of total weight not exceeding  $E$  units of weight and total volume not exceeding  $T$  units of volume, with item  $i$  having a weight of  $e_i$  integer units of weight and  $t_i$  integer units of volume.
- For each triplet  $(e, t, i)$  such that  $e \leq E$ ,  $t \leq T$ ,  $i \leq N$  we solve the following subproblem: *Choose a sub-collection of items  $1 \dots i$  that fits in a knapsack of capacity  $e$  units of weight and  $t$  units of volume, which is of largest possible value.*

- We put such a value in a 3D table of size  $E \times T \times N$  where  $N$  is the number of items (in this case jobs).
- These values are obtained using the following recursion:

$$\text{opt}(i, e, t) = \max\{\text{opt}(i - 1, e, t), \text{opt}(i - 1, e - e_i, t - t_i) + d_i\}.$$

- The complexity is  $O(NET)$

- We put such a value in a 3D table of size  $E \times T \times N$  where  $N$  is the number of items (in this case jobs).
- These values are obtained using the following recursion:

$$\text{opt}(i, e, t) = \max\{\text{opt}(i - 1, e, t), \text{opt}(i - 1, e - e_i, t - t_i) + d_i\}.$$

- The complexity is  $O(NET)$

- We put such a value in a 3D table of size  $E \times T \times N$  where  $N$  is the number of items (in this case jobs).
- These values are obtained using the following recursion:

$$\text{opt}(i, e, t) = \max\{\text{opt}(i - 1, e, t), \text{opt}(i - 1, e - e_i, t - t_i) + d_i\}.$$

- The complexity is  $O(NET)$

11. Because of the recent droughts,  $n$  proposals have been made to dam the Murray Darling river. The  $i^{th}$  proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). What is the largest number of dams that can be built? You may assume that  $x_i < x_{i+1}$ .

### Solution:

- We solve this by finding the maximum value among the following subproblems for every  $i \leq n$ : *Find the largest possible number of dams that can be built among proposals  $1 \dots i$ , such that the  $i^{th}$  dam is built.*
- The base case is  $\text{opt}(1) = 1$ .
- The recursion is:

$$\text{opt}(i) = 1 + \max\{\text{opt}(j) : x_i - x_j > \max(r_i, r_j), j < i\}$$

- We do an  $O(n)$  search at every dam proposal  $i$ . Thus, the total complexity is  $O(n^2)$ .

11. Because of the recent droughts,  $n$  proposals have been made to dam the Murray Darling river. The  $i^{th}$  proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). What is the largest number of dams that can be built? You may assume that  $x_i < x_{i+1}$ .

### Solution:

- We solve this by finding the maximum value among the following subproblems for every  $i \leq n$ : *Find the largest possible number of dams that can be built among proposals  $1 \dots i$ , such that the  $i^{th}$  dam is built.*
- The base case is  $\text{opt}(1) = 1$ .
- The recursion is:

$$\text{opt}(i) = 1 + \max\{\text{opt}(j) : x_i - x_j > \max(r_i, r_j), j < i\}$$

- We do an  $O(n)$  search at every dam proposal  $i$ . Thus, the total complexity is  $O(n^2)$ .

11. Because of the recent droughts,  $n$  proposals have been made to dam the Murray Darling river. The  $i^{th}$  proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). What is the largest number of dams that can be built? You may assume that  $x_i < x_{i+1}$ .

### Solution:

- We solve this by finding the maximum value among the following subproblems for every  $i \leq n$ : *Find the largest possible number of dams that can be built among proposals  $1 \dots i$ , such that the  $i^{th}$  dam is built.*
- The base case is  $\text{opt}(1) = 1$ .
- The recursion is:

$$\text{opt}(i) = 1 + \max\{\text{opt}(j) : x_i - x_j > \max(r_i, r_j), j < i\}$$

- We do an  $O(n)$  search at every dam proposal  $i$ . Thus, the total complexity is  $O(n^2)$ .

11. Because of the recent droughts,  $n$  proposals have been made to dam the Murray Darling river. The  $i^{th}$  proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). What is the largest number of dams that can be built? You may assume that  $x_i < x_{i+1}$ .

### Solution:

- We solve this by finding the maximum value among the following subproblems for every  $i \leq n$ : *Find the largest possible number of dams that can be built among proposals  $1 \dots i$ , such that the  $i^{th}$  dam is built.*
- The base case is  $\text{opt}(1) = 1$ .
- The recursion is:

$$\text{opt}(i) = 1 + \max\{\text{opt}(j) : x_i - x_j > \max(r_i, r_j), j < i\}$$

- We do an  $O(n)$  search at every dam proposal  $i$ . Thus, the total complexity is  $O(n^2)$ .

11. Because of the recent droughts,  $n$  proposals have been made to dam the Murray Darling river. The  $i^{th}$  proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). What is the largest number of dams that can be built? You may assume that  $x_i < x_{i+1}$ .

### Solution:

- We solve this by finding the maximum value among the following subproblems for every  $i \leq n$ : *Find the largest possible number of dams that can be built among proposals  $1 \dots i$ , such that the  $i^{th}$  dam is built.*
- The base case is  $\text{opt}(1) = 1$ .
- The recursion is:

$$\text{opt}(i) = 1 + \max\{\text{opt}(j) : x_i - x_j > \max(r_i, r_j), j < i\}$$

- We do an  $O(n)$  search at every dam proposal  $i$ . Thus, the total complexity is  $O(n^2)$ .

12. You are given an  $n \times n$  chessboard with an integer in each of its  $n^2$  squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.

- a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.
- b) Describe an algorithm which always correctly finds a minimal sum path and runs in time  $n^2$ .
- c) Describe an algorithm which computes the number of such minimal paths.
- d) (Very Tricky!) Assume now that such a chessboard is stored in a read only memory. Describe an algorithm which always correctly finds a minimal sum path and runs in linear space (i.e., amount of read/write memory used is  $O(n)$ ) and in time  $O(n^2)$ .

12. You are given an  $n \times n$  chessboard with an integer in each of its  $n^2$  squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.

- a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.
- b) Describe an algorithm which always correctly finds a minimal sum path and runs in time  $n^2$ .
- c) Describe an algorithm which computes the number of such minimal paths.
- d) (Very Tricky!) Assume now that such a chessboard is stored in a read only memory. Describe an algorithm which always correctly finds a minimal sum path and runs in linear space (i.e., amount of read/write memory used is  $O(n)$ ) and in time  $O(n^2)$ .

12. You are given an  $n \times n$  chessboard with an integer in each of its  $n^2$  squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.

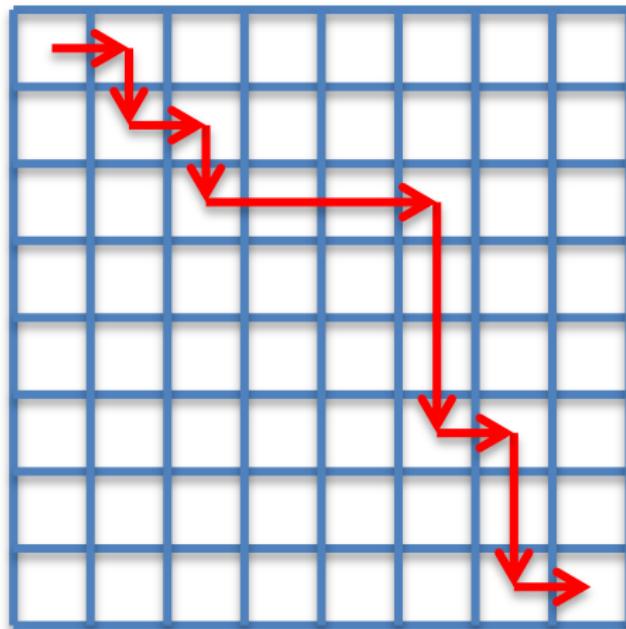
- a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.
- b) Describe an algorithm which always correctly finds a minimal sum path and runs in time  $n^2$ .
- c) Describe an algorithm which computes the number of such minimal paths.
- d) (Very Tricky!) Assume now that such a chessboard is stored in a read only memory. Describe an algorithm which always correctly finds a minimal sum path and runs in linear space (i.e., amount of read/write memory used is  $O(n)$ ) and in time  $O(n^2)$ .

12. You are given an  $n \times n$  chessboard with an integer in each of its  $n^2$  squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.

- a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.
- b) Describe an algorithm which always correctly finds a minimal sum path and runs in time  $n^2$ .
- c) Describe an algorithm which computes the number of such minimal paths.
- d) (Very Tricky!) Assume now that such a chessboard is stored in a read only memory. Describe an algorithm which always correctly finds a minimal sum path and runs in linear space (i.e., amount of read/write memory used is  $O(n)$ ) and in time  $O(n^2)$ .

$n$

$m$



**Solution: a)** Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form “*What is the best score we can get arriving at the cell at row  $i$  and column  $j$ ?*” The base case is that  $\text{opt}(1, 1) = \text{board}(1, 1)$ , and  $\text{opt}(i, j) = \infty$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

The complexity is  $O(n^2)$ .

**Solution:** a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form “*What is the best score we can get arriving at the cell at row  $i$  and column  $j$ ?*” The base case is that  $\text{opt}(1, 1) = \text{board}(1, 1)$ , and  $\text{opt}(i, j) = \infty$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

The complexity is  $O(n^2)$ .

**Solution:** a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form “*What is the best score we can get arriving at the cell at row  $i$  and column  $j$ ?*” The base case is that  $\text{opt}(1, 1) = \text{board}(1, 1)$ , and  $\text{opt}(i, j) = \infty$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

The complexity is  $O(n^2)$ .

**Solution: a)** Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm will get a score of 1002, instead of the correct answer of 15.

**b)** We solve all subproblems of the form “*What is the best score we can get arriving at the cell at row  $i$  and column  $j$ ?*” The base case is that  $\text{opt}(1, 1) = \text{board}(1, 1)$ , and  $\text{opt}(i, j) = \infty$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

The complexity is  $O(n^2)$ .

**Solution:** a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form “*What is the best score we can get arriving at the cell at row  $i$  and column  $j$ ?*” The base case is that  $\text{opt}(1, 1) = \text{board}(1, 1)$ , and  $\text{opt}(i, j) = \infty$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

The complexity is  $O(n^2)$ .

c) Solve the following subproblem: What is the minimum number of ways to reach row  $i$  and column  $j$  with the score  $\text{opt}(i, j)$ ? The base case is  $\text{ways}(1, 1) = 1$  and  $\text{ways}(i, j) = 0$  for all  $i$  and  $j$  that are off the board. The recursion is:

$$\text{ways}(i, j) = \begin{cases} \text{ways}(i - 1, j) & \text{if } \text{opt}(i - 1, j) < \text{opt}(i, j - 1) \\ \text{ways}(i, j - 1) & \text{if } \text{opt}(i - 1, j) > \text{opt}(i, j - 1) \\ \text{ways}(i - 1, j) + \text{ways}(i, j - 1) & \text{if } \text{opt}(i - 1, j) = \text{opt}(i, j - 1) \end{cases}$$

The complexity is once again  $O(n^2)$

- **d)** This is a very tricky one. The idea is to combine divide and conquer with dynamic programming.
- Note that to generate optimal scores at a row  $i$  you only need the optimal scores of the previous row.
- You start running the previous algorithm from the top left cell to the middle row  $\lfloor n/2 \rfloor$ , keeping in memory only the previous row.
- You now run the algorithm from the bottom right corner until you reach the middle row, always going either up one cell or to the left one cell.

- **d)** This is a very tricky one. The idea is to combine divide and conquer with dynamic programming.
- Note that to generate optimal scores at a row  $i$  you only need the optimal scores of the previous row.
- You start running the previous algorithm from the top left cell to the middle row  $\lfloor n/2 \rfloor$ , keeping in memory only the previous row.
- You now run the algorithm from the bottom right corner until you reach the middle row, always going either up one cell or to the left one cell.

- d) This is a very tricky one. The idea is to combine divide and conquer with dynamic programming.
- Note that to generate optimal scores at a row  $i$  you only need the optimal scores of the previous row.
- You start running the previous algorithm from the top left cell to the middle row  $\lfloor n/2 \rfloor$ , keeping in memory only the previous row.
- You now run the algorithm from the bottom right corner until you reach the middle row, always going either up one cell or to the left one cell.

- d) This is a very tricky one. The idea is to combine divide and conquer with dynamic programming.
- Note that to generate optimal scores at a row  $i$  you only need the optimal scores of the previous row.
- You start running the previous algorithm from the top left cell to the middle row  $\lfloor n/2 \rfloor$ , keeping in memory only the previous row.
- You now run the algorithm from the bottom right corner until you reach the middle row, always going either up one cell or to the left one cell.

- Once you reach the middle row you sum up the scores obtained by moving down and to the right from the top left cell and the scores obtained by moving up and to the left from the bottom right cell and you pick the cell  $C(n/2, m)$  in that row with the minimal sum.
- This clearly is the cell on the optimal trajectory.
- You now store the coordinates of that cell and proceed with the same strategy applied to the top left region for which  $C(n/2, m)$  is bottom right cell, and also applied to the bottom right region of the board for which  $C(n/2, m)$  is the top left cell.
- The run time is

$$O(n \times n + n \times n/2 + n \times n/4 + \dots) = O(n \times 2n) = O(n^2)$$

- Once you reach the middle row you sum up the scores obtained by moving down and to the right from the top left cell and the scores obtained by moving up and to the left from the bottom right cell and you pick the cell  $C(n/2, m)$  in that row with the minimal sum.
- This clearly is the cell on the optimal trajectory.
- You now store the coordinates of that cell and proceed with the same strategy applied to the top left region for which  $C(n/2, m)$  is bottom right cell, and also applied to the bottom right region of the board for which  $C(n/2, m)$  is the top left cell.
- The run time is

$$O(n \times n + n \times n/2 + n \times n/4 + \dots) = O(n \times 2n) = O(n^2)$$

- Once you reach the middle row you sum up the scores obtained by moving down and to the right from the top left cell and the scores obtained by moving up and to the left from the bottom right cell and you pick the cell  $C(n/2, m)$  in that row with the minimal sum.
- This clearly is the cell on the optimal trajectory.
- You now store the coordinates of that cell and proceed with the same strategy applied to the top left region for which  $C(n/2, m)$  is bottom right cell, and also applied to the bottom right region of the board for which  $C(n/2, m)$  is the top left cell.
- The run time is

$$O(n \times n + n \times n/2 + n \times n/4 + \dots) = O(n \times 2n) = O(n^2)$$

- Once you reach the middle row you sum up the scores obtained by moving down and to the right from the top left cell and the scores obtained by moving up and to the left from the bottom right cell and you pick the cell  $C(n/2, m)$  in that row with the minimal sum.
- This clearly is the cell on the optimal trajectory.
- You now store the coordinates of that cell and proceed with the same strategy applied to the top left region for which  $C(n/2, m)$  is bottom right cell, and also applied to the bottom right region of the board for which  $C(n/2, m)$  is the top left cell.
- The run time is

$$O(n \times n + n \times n/2 + n \times n/4 + \dots) = O(n \times 2n) = O(n^2)$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

### Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

### Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

### Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

### Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

### Solution:

- Let  $S_i$  denote the  $i^{th}$  letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter  $i$  and ending at  $j$ .*
- Subproblems are solved in order of  $j - i$ .
- The base case is that  $\text{opt}(i, i) = 1$ , as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i, j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ \text{opt}(i + 1, j - 1) + 2 & \text{if } S_i = S_j \text{ and } i < j - 2, \\ \max\{\text{opt}(i, j - 1), \text{opt}(i + 1, j)\} & \text{else} \end{cases}$$

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i, j)] = \begin{cases} (i, i) & \text{if } i = j, \\ (i, j) & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ (i + 1, j - 1) & \text{if } S_i = S_j \text{ and } i < j - 2, \\ (i, j - 1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i, j - 1) \geq \text{opt}(i + 1, j) \\ (i + 1, j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function  $\text{pred}[(i, j)]$  starting with  $(1, n)$  and recoding values of  $i$  and  $j$  when  $S_i = S_j$ .
- The complexity of this algorithm is  $O(n^2)$ , where  $n$  is the length of the string.

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i, j)] = \begin{cases} (i, i) & \text{if } i = j, \\ (i, j) & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ (i + 1, j - 1) & \text{if } S_i = S_j \text{ and } i < j - 2, \\ (i, j - 1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i, j - 1) \geq \text{opt}(i + 1, j) \\ (i + 1, j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function  $\text{pred}[(i, j)]$  starting with  $(1, n)$  and recoding values of  $i$  and  $j$  when  $S_i = S_j$ .
- The complexity of this algorithm is  $O(n^2)$ , where  $n$  is the length of the string.

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i, j)] = \begin{cases} (i, i) & \text{if } i = j, \\ (i, j) & \text{if } i + 1 = j \text{ and } S_i = S_j, \\ (i + 1, j - 1) & \text{if } S_i = S_j \text{ and } i < j - 2, \\ (i, j - 1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i, j - 1) \geq \text{opt}(i + 1, j) \\ (i + 1, j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function  $\text{pred}[(i, j)]$  starting with  $(1, n)$  and recoding values of  $i$  and  $j$  when  $S_i = S_j$ .
- The complexity of this algorithm is  $O(n^2)$ , where  $n$  is the length of the string.

14. A partition of a number  $n$  is a sequence  $\langle p_1, p_2, \dots, p_t \rangle$  (we call the  $p_k$  parts) such that  $1 \leq p_1 \leq p_2 \leq \dots \leq p_t \leq n$  and such that  $p_1 + \dots + p_t = n$ .

- ① Find the number of partitions of  $n$  in which every part is smaller or equal than  $k$ , where  $n, k$  are two given numbers such that  $1 \leq k \leq n$ .
- ② Find the total number of partitions of  $n$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

## Solution:

- (a) Let  $\text{numpart}(k, m)$  denote the number of partitions of  $m$  in which every part is at most  $k$ .
- Then for all  $m$   $\text{numpart}(1, m) = 1$  because the only way to represent  $m$  with  $k = 1$  would be as a sum of  $m$  ones.
- For  $k \geq 2$  the value of  $\text{numpart}(k, m)$  satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of  $m$  when no part is equal  $k$  and the second term counts the number of partitions when there exists at least one part equal to  $k$ , which we subtract from  $m$  and look at the number of partitions of the remainder  $m - k$  in which all the parts are also at most  $k$ .
- Note that  $\text{numpart}(k, m)$  are computed in order of the value of the sum  $m + k$ .
- (b) We run the previous algorithm; the solution is  $\text{numpart}(n, n)$ .

15. We say that a sequence of Roman letters  $A$  occurs as a subsequence of a sequence of Roman letters  $B$  if we can obtain  $A$  by deleting some of the symbols of  $B$ . Design an algorithm which for every two sequences  $A$  and  $B$  gives the number of different occurrences of  $A$  in  $B$ , i.e., the number of ways one can delete some of the symbols of  $B$  to get  $A$ . For example, the sequence  $ba$  has three occurrences in the sequence  $baba$ : baba, baba, baba.

**Solution:**

- Let  $A_i$  be the  $i^{th}$  letter of  $A$ , and  $B_j$  be the  $j^{th}$  letter of  $B$ .
- Solve the subproblems: *How many times do the first  $i$  letters of  $A$  appear as a subsequence of the first  $j$  letters of  $B$ .*
- The base case is  $\text{ways}(1, 1) = 1$  if  $A_1 = B_1$  and 0 otherwise; also, clearly,  $\text{ways}(i, j) = 0$  if  $i > j$ .

15. We say that a sequence of Roman letters  $A$  occurs as a subsequence of a sequence of Roman letters  $B$  if we can obtain  $A$  by deleting some of the symbols of  $B$ . Design an algorithm which for every two sequences  $A$  and  $B$  gives the number of different occurrences of  $A$  in  $B$ , i.e., the number of ways one can delete some of the symbols of  $B$  to get  $A$ . For example, the sequence  $ba$  has three occurrences in the sequence  $baba$ : baba, baba, baba.

### Solution:

- Let  $A_i$  be the  $i^{th}$  letter of  $A$ , and  $B_j$  be the  $j^{th}$  letter of  $B$ .
- Solve the subproblems: *How many times do the first  $i$  letters of  $A$  appear as a subsequence of the first  $j$  letters of  $B$ .*
- The base case is  $\text{ways}(1, 1) = 1$  if  $A_1 = B_1$  and 0 otherwise; also, clearly,  $\text{ways}(i, j) = 0$  if  $i > j$ .

15. We say that a sequence of Roman letters  $A$  occurs as a subsequence of a sequence of Roman letters  $B$  if we can obtain  $A$  by deleting some of the symbols of  $B$ . Design an algorithm which for every two sequences  $A$  and  $B$  gives the number of different occurrences of  $A$  in  $B$ , i.e., the number of ways one can delete some of the symbols of  $B$  to get  $A$ . For example, the sequence  $ba$  has three occurrences in the sequence  $baba$ : baba, baba, baba.

### Solution:

- Let  $A_i$  be the  $i^{th}$  letter of  $A$ , and  $B_j$  be the  $j^{th}$  letter of  $B$ .
- Solve the subproblems: *How many times do the first  $i$  letters of  $A$  appear as a subsequence of the first  $j$  letters of  $B$ .*
- The base case is  $\text{ways}(1, 1) = 1$  if  $A_1 = B_1$  and 0 otherwise; also, clearly,  $\text{ways}(i, j) = 0$  if  $i > j$ .

15. We say that a sequence of Roman letters  $A$  occurs as a subsequence of a sequence of Roman letters  $B$  if we can obtain  $A$  by deleting some of the symbols of  $B$ . Design an algorithm which for every two sequences  $A$  and  $B$  gives the number of different occurrences of  $A$  in  $B$ , i.e., the number of ways one can delete some of the symbols of  $B$  to get  $A$ . For example, the sequence  $ba$  has three occurrences in the sequence  $baba$ : baba, baba, baba.

### Solution:

- Let  $A_i$  be the  $i^{th}$  letter of  $A$ , and  $B_j$  be the  $j^{th}$  letter of  $B$ .
- Solve the subproblems: *How many times do the first  $i$  letters of  $A$  appear as a subsequence of the first  $j$  letters of  $B$ .*
- The base case is  $\text{ways}(1, 1) = 1$  if  $A_1 = B_1$  and 0 otherwise; also, clearly,  $\text{ways}(i, j) = 0$  if  $i > j$ .

The recursion is, for all  $i < j$

$$\text{opt}(i, j) = \begin{cases} 0, & \text{if } i > j \\ \text{opt}(i, j - 1), & \text{if } A_i \neq B_j \\ \text{opt}(i - 1, j - 1) + \text{opt}(i, j - 1) & \text{if } A_i = B_j \end{cases}$$

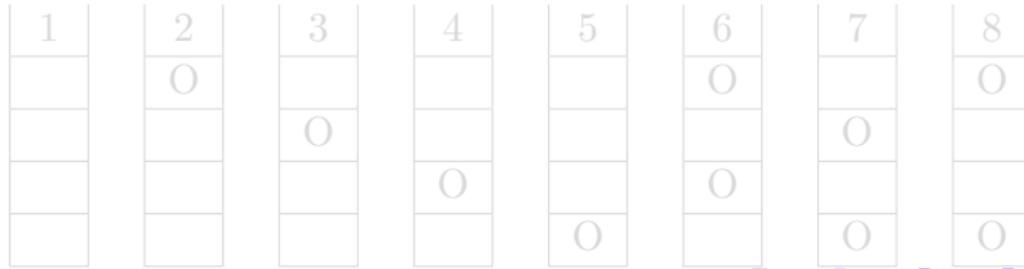
Note that in the second case when  $A_i = B_j$  we have two options: we can map  $A_i$  into  $B_j$  because they match, but we can also map the first  $i$  many letters of  $A$  into  $j - 1$  many letters of  $B$ , so we have the sum of these two options.

The complexity is  $O(nm)$  where  $n$  is the length of  $A$  and  $m$  is the length of  $B$ .

16. We are given a checkerboard which has 4 rows and  $n$  columns, and has an integer written in each square. We are also given a set of  $2n$  pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximise the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine). Give an  $O(n)$ -time algorithm for computing an optimal placement.

**Solution:**

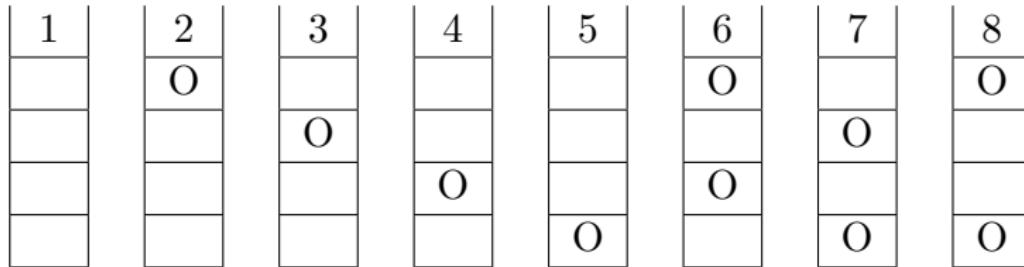
- There are 8 patterns, listed below, which are legal and that can occur in any column (in isolation, ignoring the pebbles in adjacent columns):



16. We are given a checkerboard which has 4 rows and  $n$  columns, and has an integer written in each square. We are also given a set of  $2n$  pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximise the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine). Give an  $O(n)$ -time algorithm for computing an optimal placement.

### Solution:

- There are 8 patterns, listed below, which are legal and that can occur in any column (in isolation, ignoring the pebbles in adjacent columns):



- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
  - We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
  - The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
  - The base case is that  $\text{opt}(0) = 0$ .
  - The recursion is:
$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$
  - Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{\text{th}}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{th}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{th}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement.
- Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ .
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- Let  $t$  denote the type of pattern, so that  $t$  ranges from 1 to 8.
- We solve our subproblem by choosing  $k$  columns from our set of patterns, such that each column is compatible with the one before.
- The subproblem is: *What is the maximum score we can get by only placing pebbles in the first  $k$  columns, such the  $k^{th}$  column has pattern  $t$ .*
- The base case is that  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(k, t) = \text{score}(k, t) + \max\{\text{opt}(k - 1, s) : s \text{ is compatible with } t\}$$

- Here,  $\text{score}(k, t)$  is the score obtained by using pattern  $t$  on column  $k$ .

- The compatibilities are as follows:
  - (a) pattern 1 is compatible with all 8 patterns (including itself);
  - (b) pattern 2 is compatible with all patterns except 2,6 and 8
  - (c) pattern 3 is compatible with all patterns except 3 and 7;
  - (d) pattern 4 is compatible with all patterns except 4 and 6;
  - (e) pattern 5 is compatible with all patterns except 5,7 and 8;
  - (f) pattern 6 is compatible with all patterns except 2,6 and 8;
  - (g) pattern 7 is compatible with all patterns except 3,5,7 and 8;
  - (h) pattern 8 is compatible with all patterns except 2,5,6,7 and 8.
- The complexity is  $O(n)$ , as the number of patterns is constant.

- The compatibilities are as follows:
  - (a) pattern 1 is compatible with all 8 patterns (including itself);
  - (b) pattern 2 is compatible with all patterns except 2,6 and 8
  - (c) pattern 3 is compatible with all patterns except 3 and 7;
  - (d) pattern 4 is compatible with all patterns except 4 and 6;
  - (e) pattern 5 is compatible with all patterns except 5,7 and 8;
  - (f) pattern 6 is compatible with all patterns except 2,6 and 8;
  - (g) pattern 7 is compatible with all patterns except 3,5,7 and 8;
  - (h) pattern 8 is compatible with all patterns except 2,5,6,7 and 8.
- The complexity is  $O(n)$ , as the number of patterns is constant.

17. Skiers go fastest with skis whose length is about their height. Your team consists of  $n$  members, with heights  $h_1, h_2, \dots, h_n$ . Your team gets a delivery of  $m \geq n$  pairs of skis, with lengths  $l_1, l_2, \dots, l_m$ . Your goal is to design an algorithm to assign to each skier one pair of skis to minimise the sum of the absolute differences between the height  $h_i$  of the skier and the length of the corresponding ski he got, i.e., to minimise

$$\sum_{1 \leq i \leq n} |h_i - l_{s(i)}|$$

where  $s(i)$  is the index of the skies assigned to the skier of height  $h_i$ .

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- First observe that if we have two skiers  $i$  and  $j$  such that  $h_i < h_j$ , then  $s(i) < s(j)$ . If this were not the case, we could swap the skis assigned to  $i$  and  $j$ , which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give  $i^{th}$  skier  $i^{th}$  pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first  $i$  skiers with the first  $j > i$  skis such that each of the first  $i$  skiers gets a ski?*
- The base case is  $\text{opt}(i, i) = \sum_{k=1}^i |h_k - l_k|$ .
- The recursion for  $j > i$  is:

$$\text{opt}(i, j) = \min\{\text{opt}(i, j - 1), \text{opt}(i - 1, j - 1) + |h_i - l_j|\}.$$

- To retrieve the assignment, we start at  $(i, j)$  where  $i = n$  and  $j = m$ . If  $\text{opt}(i - 1, j - 1) + |h_i - l_j| < \text{opt}(i, j - 1)$  then  $s(i) = j$  and we try again at  $(i - 1, j - 1)$ .
- Otherwise, we try again at  $(i, j - 1)$ .
- If at any point we reach  $i = j$  (our base case), we simply assign  $s(k) = k$  for all  $1 \leq k \leq i$ .
- The complexity of the initial recursion  $O(nm)$ . The complexity of retrieving the assignment is  $O(m)$ , as each time we run the algorithm,  $j$  decreases by exactly 1.

- To retrieve the assignment, we start at  $(i, j)$  where  $i = n$  and  $j = m$ . If  $\text{opt}(i - 1, j - 1) + |h_i - l_j| < \text{opt}(i, j - 1)$  then  $s(i) = j$  and we try again at  $(i - 1, j - 1)$ .
- Otherwise, we try again at  $(i, j - 1)$ .
- If at any point we reach  $i = j$  (our base case), we simply assign  $s(k) = k$  for all  $1 \leq k \leq i$ .
- The complexity of the initial recursion  $O(nm)$ . The complexity of retrieving the assignment is  $O(m)$ , as each time we run the algorithm,  $j$  decreases by exactly 1.

- To retrieve the assignment, we start at  $(i, j)$  where  $i = n$  and  $j = m$ . If  $\text{opt}(i - 1, j - 1) + |h_i - l_j| < \text{opt}(i, j - 1)$  then  $s(i) = j$  and we try again at  $(i - 1, j - 1)$ .
- Otherwise, we try again at  $(i, j - 1)$ .
- If at any point we reach  $i = j$  (our base case), we simply assign  $s(k) = k$  for all  $1 \leq k \leq i$ .
- The complexity of the initial recursion  $O(nm)$ . The complexity of retrieving the assignment is  $O(m)$ , as each time we run the algorithm,  $j$  decreases by exactly 1.

- To retrieve the assignment, we start at  $(i, j)$  where  $i = n$  and  $j = m$ . If  $\text{opt}(i - 1, j - 1) + |h_i - l_j| < \text{opt}(i, j - 1)$  then  $s(i) = j$  and we try again at  $(i - 1, j - 1)$ .
- Otherwise, we try again at  $(i, j - 1)$ .
- If at any point we reach  $i = j$  (our base case), we simply assign  $s(k) = k$  for all  $1 \leq k \leq i$ .
- The complexity of the initial recursion  $O(nm)$ . The complexity of retrieving the assignment is  $O(m)$ , as each time we run the algorithm,  $j$  decreases by exactly 1.

18. You have to cut a wood stick into several pieces. The most affordable company, Analog Cutting Machinery (ACM), charges money according to the length of the stick being cut. Their cutting saw allows them to make only one cut at a time. It is easy to see that different cutting orders can lead to different prices.

For example, consider a stick of length 10 m that has to be cut at 2, 4, and 7 m from one end. There are several choices. One can cut first at 2, then at 4, then at 7. This leads to a price of  $10 + 8 + 6 = 24$  because the first stick was of 10 m, the resulting stick of 8 m, and the last one of 6 m. Another choice could cut at 4, then at 2, then at 7. This would lead to a price of  $10 + 4 + 6 = 20$ , which is better for us.

Your boss demands that you design an algorithm to find the minimum possible cutting cost for any given stick.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let  $x(i)$  be the distance along the stick the  $i$ th cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up the stick completely from cutting point  $i$  to cutting point  $j$ .*
- The base case is  $\text{opt}(i, i + 1) = 0$ .
- The recursion is:

$$\text{opt}(i, j) = x(j) - x(i) + \min\{\text{opt}(i, k) + \text{opt}(k, j) : i < k < j\}.$$

- The complexity is  $O(n^3)$ , where  $n$  is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

**Solution:**

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

### Solution:

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

### Solution:

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

### Solution:

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

### Solution:

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

19. For bit strings  $X = x_1 \dots x_m$ ,  $Y = y_1 \dots y_n$  and  $Z = z_1 \dots z_{m+n}$  we say that  $Z$  is an interleaving of  $X$  and  $Y$  if it can be obtained by interleaving the bits in  $X$  and  $Y$  in a way that maintains the left-to-right order of the bits in  $X$  and  $Y$ .

For example if  $X = 101$  and  $Y = 01$  then  $x_1x_2y_1x_3y_2 = 10011$  is an interleaving of  $X$  and  $Y$ , whereas  $11010$  is not. Give an efficient algorithm to determine if  $Z$  is an interleaving of  $X$  and  $Y$ .

### Solution:

- Solve the following subproblems: *Do the first  $i$  bits of  $X$  and the first  $j$  bits of  $Y$  form an interleaving in the first  $i + j$  bits of  $Z$ ?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ , and  $\text{opt}(i, j) = \text{false}$  if  $i < 0$  or  $j < 0$ .
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of  $i + j$ .
- The complexity is  $O(nm)$ .

20. Some people think that the bigger an elephant is, the smarter it is. To disprove this you want to analyse a collection of elephants and place as large a subset of elephants as possible into a sequence whose weights are increasing but their IQs are decreasing. Design an algorithm which given the weights and IQs of  $n$  elephants, will find a longest sequence of elephants such that their weights are increasing but IQs are decreasing.

**Solution:** Sort the elephants in order of decreasing IQs. The problem now becomes finding the longest increasing subsequence of elephant weights.

21. You have been handed responsibility for a business in Texas for the next  $N$  days. Initially, you have  $K$  illegal workers. At the beginning of each day, you may hire an illegal worker, keep the number of illegal workers the same or fire an illegal worker. At the end of each day, there will be an inspection. The inspector on the  $i^{th}$  day will check that you have between  $l_i$  and  $r_i$  illegal workers (inclusive). If you do not, you will fail the inspection. Design an algorithm that determines the fewest number of inspections you will fail if you hire and fire illegal employees optimally.

**Solution:** Observe that the minimum and the maximum number of illegal workers you could possibly have on the evening of day  $i$  are  $\max(K - i, 0)$  and  $K + i$ .

We solve the following subproblems: *What is the minimum number of inspections I have failed on day  $i$  assuming that on that day I have  $j$  many illegal workers?*

21. You have been handed responsibility for a business in Texas for the next  $N$  days. Initially, you have  $K$  illegal workers. At the beginning of each day, you may hire an illegal worker, keep the number of illegal workers the same or fire an illegal worker. At the end of each day, there will be an inspection. The inspector on the  $i^{th}$  day will check that you have between  $l_i$  and  $r_i$  illegal workers (inclusive). If you do not, you will fail the inspection. Design an algorithm that determines the fewest number of inspections you will fail if you hire and fire illegal employees optimally.

**Solution:** Observe that the minimum and the maximum number of illegal workers you could possibly have on the evening of day  $i$  are  $\max(K - i, 0)$  and  $K + i$ .

We solve the following subproblems: *What is the minimum number of inspections I have failed on day  $i$  assuming that on that day I have  $j$  many illegal workers?*

- The base case is  $\text{opt}(0, K) = 0$ , since we start with 0 failed inspections before the first day begins.
- Let  $\text{failed}(i, j)$  return 1 if the  $j$  falls out of the range of  $[l_i, r_i]$ , and return 0 otherwise.
- The recursion is: for all  $i$  such that  $1 \leq i \leq N$  and all  $j$  such that  $\max(0, K - i) \leq j \leq K + i$ ,

- The base case is  $\text{opt}(0, K) = 0$ , since we start with 0 failed inspections before the first day begins.
- Let  $\text{failed}(i, j)$  return 1 if the  $j$  falls out of the range of  $[l_i, r_i]$ , and return 0 otherwise.
- The recursion is: for all  $i$  such that  $1 \leq i \leq N$  and all  $j$  such that  $\max(0, K - i) \leq j \leq K + i$ ,

- The base case is  $\text{opt}(0, K) = 0$ , since we start with 0 failed inspections before the first day begins.
- Let  $\text{failed}(i, j)$  return 1 if the  $j$  falls out of the range of  $[l_i, r_i]$ , and return 0 otherwise.
- The recursion is: for all  $i$  such that  $1 \leq i \leq N$  and all  $j$  such that  $\max(0, K - i) \leq j \leq K + i$ ,

$$\text{opt}(i, j) = \begin{cases} \begin{cases} \text{opt}(i - 1, j - 1) & \text{if } j - 1 \geq \max(0, K - (i - 1)) \\ \infty & \text{if } j - 1 < \max(0, K - (i - 1)) \end{cases} \\ \text{failed}(i, j) + \min \begin{cases} \text{opt}(i - 1, j), \\ \begin{cases} \text{opt}(i - 1, j + 1) & \text{if } j + 1 \leq K + i - 1 \\ \infty & \text{if } j + 1 > K + i - 1 \end{cases} \end{cases} \end{cases}$$

- Note that the first option in the cases corresponds to hiring an illegal worker on the morning of day  $i$ , providing that the number of workers on the previous day was achievable over the period of  $i - 1$  days (you start with  $K$  workers, so after  $i - 1$  days you must have at least  $K - (i - 1)$  workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i, j) = \begin{cases} \begin{cases} \text{opt}(i - 1, j - 1) & \text{if } j - 1 \geq \max(0, K - (i - 1)) \\ \infty & \text{if } j - 1 < \max(0, K - (i - 1)) \end{cases} \\ \text{failed}(i, j) + \min \begin{cases} \text{opt}(i - 1, j), \\ \begin{cases} \text{opt}(i - 1, j + 1) & \text{if } j + 1 \leq K + i - 1 \\ \infty & \text{if } j + 1 > K + i - 1 \end{cases} \end{cases} \end{cases}$$

- Note that the first option in the cases corresponds to hiring an illegal worker on the morning of day  $i$ , providing that the number of workers on the previous day was achievable over the period of  $i - 1$  days (you start with  $K$  workers, so after  $i - 1$  days you must have at least  $K - (i - 1)$  workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i, j) = \begin{cases} \begin{cases} \text{opt}(i - 1, j - 1) & \text{if } j - 1 \geq \max(0, K - (i - 1)) \\ \infty & \text{if } j - 1 < \max(0, K - (i - 1)) \end{cases} \\ \text{failed}(i, j) + \min \begin{cases} \text{opt}(i - 1, j), \\ \begin{cases} \text{opt}(i - 1, j + 1) & \text{if } j + 1 \leq K + i - 1 \\ \infty & \text{if } j + 1 > K + i - 1 \end{cases} \end{cases} \end{cases}$$

- Note that the first option in the cases corresponds to hiring an illegal worker on the morning of day  $i$ , providing that the number of workers on the previous day was achievable over the period of  $i - 1$  days (you start with  $K$  workers, so after  $i - 1$  days you must have at least  $K - (i - 1)$  workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i, j) = \begin{cases} \begin{cases} \text{opt}(i - 1, j - 1) & \text{if } j - 1 \geq \max(0, K - (i - 1)) \\ \infty & \text{if } j - 1 < \max(0, K - (i - 1)) \end{cases} \\ \text{failed}(i, j) + \min \begin{cases} \text{opt}(i - 1, j), \\ \begin{cases} \text{opt}(i - 1, j + 1) & \text{if } j + 1 \leq K + i - 1 \\ \infty & \text{if } j + 1 > K + i - 1 \end{cases} \end{cases} \end{cases}$$

- Note that the first option in the cases corresponds to hiring an illegal worker on the morning of day  $i$ , providing that the number of workers on the previous day was achievable over the period of  $i - 1$  days (you start with  $K$  workers, so after  $i - 1$  days you must have at least  $K - (i - 1)$  workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

- The final answer is equal to  $\min_{\max(K-N,0) \leq j \leq K+N} \text{opt}(N, j)$ .
- The complexity is  $O(N^2)$ , as there are  $N$  days, and at most  $2N + 1$  possible values of illegal worker each day.

- The final answer is equal to  $\min_{\max(K-N,0) \leq j \leq K+N} \text{opt}(N, j)$ .
- The complexity is  $O(N^2)$ , as there are  $N$  days, and at most  $2N + 1$  possible values of illegal worker each day.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

**Solution:**

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

### Solution:

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

### Solution:

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

### Solution:

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

### Solution:

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

22. Given an array of  $N$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $K$ .

### Solution:

- Solve the subproblems: *What is the number of ways I can split the first  $i$  elements into contiguous blocks of sum at most  $K$ .*
- The base case is  $\text{opt}(0) = 1$ .
- For  $1 \leq j < i$  let  $\text{sum}(j, i)$  is the sum of all numbers from  $A[j]$  to  $A[i]$  inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j - 1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is  $O(n^2)$ , as each subproblem requires a  $O(n)$  search.

23. Given a 2D  $R \times C$  grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a ‘climb’. We solve the subproblems: *What is the minimum number of ‘climbs’ to reach row  $i$  and column  $j$ .*
- The base case is  $\text{opt}(0, 0) = 0$ , and  $\text{opt}(i, j) = 0$  if  $i$  and  $j$  lie off the board.
- The recursion is:

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i - 1, j) & \text{if } \text{height}(i - 1, j) \geq \text{height}(i, j) \\ \text{opt}(i - 1, j) + 1 & \text{if } \text{height}(i - 1, j) < \text{height}(i, j) \\ \text{opt}(i, j - 1) & \text{if } \text{height}(i, j - 1) \geq \text{height}(i, j) \\ \text{opt}(i, j - 1) + 1 & \text{if } \text{height}(i, j - 1) < \text{height}(i, j) \end{cases}$$

- The complexity is  $O(RC)$ .

23. Given a 2D  $R \times C$  grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

### Solution:

- Let a move from lower elevation to higher elevation be called a ‘climb’. We solve the subproblems: *What is the minimum number of ‘climbs’ to reach row  $i$  and column  $j$ .*
- The base case is  $\text{opt}(0, 0) = 0$ , and  $\text{opt}(i, j) = 0$  if  $i$  and  $j$  lie off the board.
- The recursion is:

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i - 1, j) & \text{if } \text{height}(i - 1, j) \geq \text{height}(i, j) \\ \text{opt}(i - 1, j) + 1 & \text{if } \text{height}(i - 1, j) < \text{height}(i, j) \\ \text{opt}(i, j - 1) & \text{if } \text{height}(i, j - 1) \geq \text{height}(i, j) \\ \text{opt}(i, j - 1) + 1 & \text{if } \text{height}(i, j - 1) < \text{height}(i, j) \end{cases}$$

- The complexity is  $O(RC)$ .

23. Given a 2D  $R \times C$  grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

### Solution:

- Let a move from lower elevation to higher elevation be called a ‘climb’. We solve the subproblems: *What is the minimum number of ‘climbs’ to reach row  $i$  and column  $j$ .*
- The base case is  $\text{opt}(0, 0) = 0$ , and  $\text{opt}(i, j) = 0$  if  $i$  and  $j$  lie off the board.
- The recursion is:

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i - 1, j) & \text{if } \text{height}(i - 1, j) \geq \text{height}(i, j) \\ \text{opt}(i - 1, j) + 1 & \text{if } \text{height}(i - 1, j) < \text{height}(i, j) \\ \text{opt}(i, j - 1) & \text{if } \text{height}(i, j - 1) \geq \text{height}(i, j) \\ \text{opt}(i, j - 1) + 1 & \text{if } \text{height}(i, j - 1) < \text{height}(i, j) \end{cases}$$

- The complexity is  $O(RC)$ .

23. Given a 2D  $R \times C$  grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

### Solution:

- Let a move from lower elevation to higher elevation be called a ‘climb’. We solve the subproblems: *What is the minimum number of ‘climbs’ to reach row  $i$  and column  $j$ .*
- The base case is  $\text{opt}(0, 0) = 0$ , and  $\text{opt}(i, j) = 0$  if  $i$  and  $j$  lie off the board.
- The recursion is:

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i - 1, j) & \text{if } \text{height}(i - 1, j) \geq \text{height}(i, j) \\ \text{opt}(i - 1, j) + 1 & \text{if } \text{height}(i - 1, j) < \text{height}(i, j) \\ \text{opt}(i, j - 1) & \text{if } \text{height}(i, j - 1) \geq \text{height}(i, j) \\ \text{opt}(i, j - 1) + 1 & \text{if } \text{height}(i, j - 1) < \text{height}(i, j) \end{cases}$$

- The complexity is  $O(RC)$ .

23. Given a 2D  $R \times C$  grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

### Solution:

- Let a move from lower elevation to higher elevation be called a ‘climb’. We solve the subproblems: *What is the minimum number of ‘climbs’ to reach row  $i$  and column  $j$ .*
- The base case is  $\text{opt}(0, 0) = 0$ , and  $\text{opt}(i, j) = 0$  if  $i$  and  $j$  lie off the board.
- The recursion is:

$$\text{opt}(i, j) = \min \begin{cases} \text{opt}(i - 1, j) & \text{if } \text{height}(i - 1, j) \geq \text{height}(i, j) \\ \text{opt}(i - 1, j) + 1 & \text{if } \text{height}(i - 1, j) < \text{height}(i, j) \\ \text{opt}(i, j - 1) & \text{if } \text{height}(i, j - 1) \geq \text{height}(i, j) \\ \text{opt}(i, j - 1) + 1 & \text{if } \text{height}(i, j - 1) < \text{height}(i, j) \end{cases}$$

- The complexity is  $O(RC)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

**Solution:**

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

### Solution:

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

### Solution:

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

### Solution:

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

### Solution:

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

24. There are  $N$  levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly  $K$  levels.

### Solution:

- The subproblems are: *Can I reach the  $i^{th}$  level after playing exactly  $k$  levels?*
- The base case is  $\text{opt}(0, 0) = \text{true}$ .
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ .
- The final answer is  $\text{opt}(K, N)$ . The time complexity is  $O(N^2)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

**Solution:**

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

**Solution:**

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

**Solution:**

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

**Solution:**

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

25. You are on vacation for  $N$  days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

**Solution:**

- We solve the following subproblems: *What is the maximum enjoyment by day  $i$  if I do activity  $j$  on that day?*
- The base case is  $\text{opt}(0, j) = 0$ .
- Let the enjoyment of activity  $j$  be  $e_j$ . The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i - 1, k) : k \neq j\}.$$

- The solution to the stated problem is  $\max_{1 \leq j \leq 3} \text{opt}(N, j)$ .
- As there are only three activities to consider, the complexity is  $O(N)$ .

26. Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer  $i$ ?*
- The base case is  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(i) = A_i + \max(0, \text{opt}(i - 1)).$$

- Here, we take the maximum of 0 and  $\text{opt}(i - 1)$  because we may either choose to add  $A_i$  to the previous block, or start a completely new block.

26. Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

**Solution:**

- We solve the subproblems: *What is the maximum sum of elements ending with integer  $i$ ?*
- The base case is  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(i) = A_i + \max(0, \text{opt}(i - 1)).$$

- Here, we take the maximum of 0 and  $\text{opt}(i - 1)$  because we may either choose to add  $A_i$  to the previous block, or start a completely new block.

26. Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

**Solution:**

- We solve the subproblems: *What is the maximum sum of elements ending with integer  $i$ ?*
- The base case is  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(i) = A_i + \max(0, \text{opt}(i - 1)).$$

- Here, we take the maximum of 0 and  $\text{opt}(i - 1)$  because we may either choose to add  $A_i$  to the previous block, or start a completely new block.

26. Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

**Solution:**

- We solve the subproblems: *What is the maximum sum of elements ending with integer  $i$ ?*
- The base case is  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(i) = A_i + \max(0, \text{opt}(i - 1)).$$

- Here, we take the maximum of 0 and  $\text{opt}(i - 1)$  because we may either choose to add  $A_i$  to the previous block, or start a completely new block.

26. Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

**Solution:**

- We solve the subproblems: *What is the maximum sum of elements ending with integer  $i$ ?*
- The base case is  $\text{opt}(0) = 0$ .
- The recursion is:

$$\text{opt}(i) = A_i + \max(0, \text{opt}(i - 1)).$$

- Here, we take the maximum of 0 and  $\text{opt}(i - 1)$  because we may either choose to add  $A_i$  to the previous block, or start a completely new block.

- To determine the actual block, we may also solve the following recursive function for all  $i$ :

$$\text{start}(i) = \begin{cases} \text{start}(i - 1) & \text{if } \text{opt}(i - 1) > 0, \\ i & \text{if } \text{opt}(i - 1) \leq 0 \end{cases}$$

- To get the block, we simply find the element  $i$  such that  $\text{opt}(i)$  is maximised. The block with maximum sum is  $[\text{start}(i), i]$ .
- The complexity is  $O(n)$ .

- To determine the actual block, we may also solve the following recursive function for all  $i$ :

$$\text{start}(i) = \begin{cases} \text{start}(i - 1) & \text{if } \text{opt}(i - 1) > 0, \\ i & \text{if } \text{opt}(i - 1) \leq 0 \end{cases}$$

- To get the block, we simply find the element  $i$  such that  $\text{opt}(i)$  is maximised. The block with maximum sum is  $[\text{start}(i), i]$ .
- The complexity is  $O(n)$ .

- To determine the actual block, we may also solve the following recursive function for all  $i$ :

$$\text{start}(i) = \begin{cases} \text{start}(i - 1) & \text{if } \text{opt}(i - 1) > 0, \\ i & \text{if } \text{opt}(i - 1) \leq 0 \end{cases}$$

- To get the block, we simply find the element  $i$  such that  $\text{opt}(i)$  is maximised. The block with maximum sum is  $[\text{start}(i), i]$ .
- The complexity is  $O(n)$ .

27. Consider a row of  $n$  coins of values  $v_1, v_2, \dots, v_n$ , where  $n$  is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

**Solution:**

- Let  $\text{opt}(i, j)$  for  $i, j$  two integers such that  $j - i + 1$  an even number and  $1 \leq i < j \leq n$  denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position  $i$  and position  $j$ .
- Then  $\text{opt}(i, j)$  satisfies the following recursion

$$\begin{aligned}\text{opt}(i, j) = & \max\{v_i + \min\{\text{opt}(i + 2, j), \text{opt}(i + 1, j - 1)\}, \\ & v_j + \min\{\text{opt}(i + 1, j - 1), \text{opt}(i, j - 2)\}\}\end{aligned}$$

27. Consider a row of  $n$  coins of values  $v_1, v_2, \dots, v_n$ , where  $n$  is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

### Solution:

- Let  $\text{opt}(i, j)$  for  $i, j$  two integers such that  $j - i + 1$  an even number and  $1 \leq i < j \leq n$  denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position  $i$  and position  $j$ .
- Then  $\text{opt}(i, j)$  satisfies the following recursion

$$\begin{aligned}\text{opt}(i, j) = & \max\{v_i + \min\{\text{opt}(i+2, j), \text{opt}(i+1, j-1)\}, \\ & v_j + \min\{\text{opt}(i+1, j-1), \text{opt}(i, j-2)\}\}\end{aligned}$$

27. Consider a row of  $n$  coins of values  $v_1, v_2, \dots, v_n$ , where  $n$  is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

### Solution:

- Let  $\text{opt}(i, j)$  for  $i, j$  two integers such that  $j - i + 1$  an even number and  $1 \leq i < j \leq n$  denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position  $i$  and position  $j$ .
- Then  $\text{opt}(i, j)$  satisfies the following recursion

$$\begin{aligned}\text{opt}(i, j) = & \max\{v_i + \min\{\text{opt}(i + 2, j), \text{opt}(i + 1, j - 1)\}, \\ & v_j + \min\{\text{opt}(i + 1, j - 1), \text{opt}(i, j - 2)\}\}\end{aligned}$$

- The options inside the min functions represent the choice your opponent takes, either to continue taking from the same end as you took or from the opposite end of the sequence.
- The problems to find  $\text{opt}(i, j)$  are solved in order of the size of  $j - i$ .
- The complexity is  $O(n^2)$ , because this is how many pairs of  $i, j$  we have and each stage of the recursion has only constant number of steps (4 table lookups and 2 additions plus two min and one max operations).

- The options inside the min functions represent the choice your opponent takes, either to continue taking from the same end as you took or from the opposite end of the sequence.
- The problems to find  $\text{opt}(i, j)$  are solved in order of the size of  $j - i$ .
- The complexity is  $O(n^2)$ , because this is how many pairs of  $i, j$  we have and each stage of the recursion has only constant number of steps (4 table lookups and 2 additions plus two min and one max operations).

- The options inside the min functions represent the choice your opponent takes, either to continue taking from the same end as you took or from the opposite end of the sequence.
- The problems to find  $\text{opt}(i, j)$  are solved in order of the size of  $j - i$ .
- The complexity is  $O(n^2)$ , because this is how many pairs of  $i, j$  we have and each stage of the recursion has only constant number of steps (4 table lookups and 2 additions plus two min and one max operations).

- (a) Multiply two complex numbers  $(a + ib)$  and  $(c + id)$  (where  $a, b, c, d$  are all real numbers) using only 3 real number multiplications.
- (b) Find  $(a + ib)^2$  using only two multiplications of real numbers.
- (c) Find the product  $(a + ib)^2(c + id)^2$  using only five real number multiplications.

**Solution:**

- (a) This is again the Karatsuba trick:

$$(a+ib)(c+id) = ac - bd + (bc + ad)i = ac - bd + ((a+b)(c+d) - ac - bd)i$$

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

- (b)  $(a + ib)^2 = a^2 - b^2 + 2abi = (a + b)(a - b) + (a + a)bi$ .
- (c) Just note that  $(a + ib)^2(c + id)^2 = ((a + ib)(c + id))^2$ ; you can now use (a) to multiply  $(a + ib)(c + id)$  with only three multiplications and then you can use (b) to square the result with two additional multiplications.

- (a) Multiply two complex numbers  $(a + ib)$  and  $(c + id)$  (where  $a, b, c, d$  are all real numbers) using only 3 real number multiplications.
- (b) Find  $(a + ib)^2$  using only two multiplications of real numbers.
- (c) Find the product  $(a + ib)^2(c + id)^2$  using only five real number multiplications.

### Solution:

- (a) This is again the Karatsuba trick:

$$(a+ib)(c+id) = ac - bd + (bc + ad)i = ac - bd + ((a+b)(c+d) - ac - bd)i$$

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

- (b)  $(a + ib)^2 = a^2 - b^2 + 2abi = (a + b)(a - b) + (a + a)bi$ .
- (c) Just note that  $(a + ib)^2(c + id)^2 = ((a + ib)(c + id))^2$ ; you can now use (a) to multiply  $(a + ib)(c + id)$  with only three multiplications and then you can use (b) to square the result with two additional multiplications.

- (a) Multiply two complex numbers  $(a + ib)$  and  $(c + id)$  (where  $a, b, c, d$  are all real numbers) using only 3 real number multiplications.
- (b) Find  $(a + ib)^2$  using only two multiplications of real numbers.
- (c) Find the product  $(a + ib)^2(c + id)^2$  using only five real number multiplications.

### Solution:

- (a) This is again the Karatsuba trick:

$$(a+ib)(c+id) = ac - bd + (bc + ad)i = ac - bd + ((a+b)(c+d) - ac - bd)i$$

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

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

- (a) Multiply two complex numbers  $(a + ib)$  and  $(c + id)$  (where  $a, b, c, d$  are all real numbers) using only 3 real number multiplications.
- (b) Find  $(a + ib)^2$  using only two multiplications of real numbers.
- (c) Find the product  $(a + ib)^2(c + id)^2$  using only five real number multiplications.

### Solution:

- (a) This is again the Karatsuba trick:

$$(a+ib)(c+id) = ac - bd + (bc + ad)i = ac - bd + ((a+b)(c+d) - ac - bd)i$$

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

- (b)  $(a + ib)^2 = a^2 - b^2 + 2ab i = (a + b)(a - b) + (a + b)bi$ .
- (c) Just note that  $(a + ib)^2(c + id)^2 = ((a + ib)(c + id))^2$ ; you can now use (a) to multiply  $(a + ib)(c + id)$  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

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

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

Let 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  $O(\log n)$ .

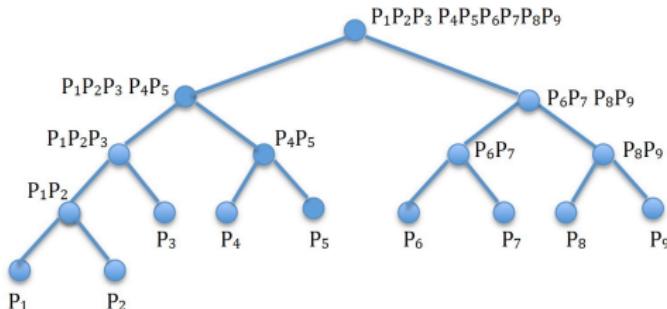
Let

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

- 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  $O(\log n)$ .

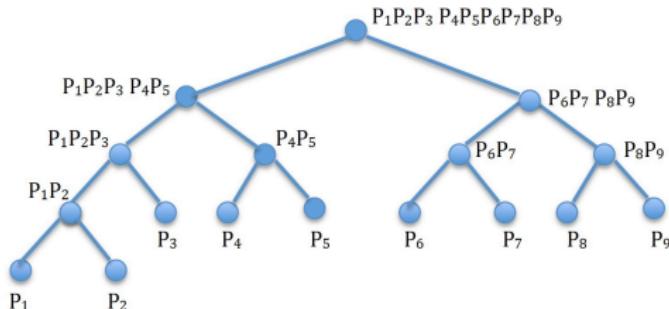
- Assume you are given  $n$  sorted arrays  $P_i$ ,  $1 \leq i \leq n$ , of different sizes  $e_i$ . You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left. Design an algorithm that achieves this task and uses minimal total number 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  $X_1, \dots, X_n$  of real numbers and an integer  $k < n$ . Construct a sequence of  $k$  intervals  $Y_1, \dots, Y_k$  which cover intervals  $X_1, \dots, X_n$  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  $X_1, \dots, X_n$  with a single interval  $Z_1$ . Then look for the largest gap and excise it from  $Z_1$ . Continue excising gaps between intervals  $X_1, \dots, X_n$  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  $X_1, \dots, X_n$  of real numbers and an integer  $k < n$ . Construct a sequence of  $k$  intervals  $Y_1, \dots, Y_k$  which cover intervals  $X_1, \dots, X_n$  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  $X_1, \dots, X_n$  with a single interval  $Z_1$ . Then look for the largest gap and excise it from  $Z_1$ . Continue excising gaps between intervals  $X_1, \dots, X_n$  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  $X_1, \dots, X_n$  of real numbers and an integer  $k < n$ . Construct a sequence of  $k$  intervals  $Y_1, \dots, Y_k$  which cover intervals  $X_1, \dots, X_n$  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  $X_1, \dots, X_n$  with a single interval  $Z_1$ . Then look for the largest gap and excise it from  $Z_1$ . Continue excising gaps between intervals  $X_1, \dots, X_n$  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  $1, c, c^2, c^3, \dots, c^n$  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  $1, c, c^2, c^3, \dots, c^n$  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  $1, c, c^2, c^3, \dots, c^n$  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  $1, c, c^2, c^3, \dots, c^n$  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  $n, A, B, a[1 \dots n]$  and  $b[1 \dots n]$ , 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  $n, A, B, a[1 \dots n]$  and  $b[1 \dots n]$ , 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  $n, A, B, a[1 \dots n]$  and  $b[1 \dots n]$ , 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  $v_1, \dots, v_n$  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  $v_1, \dots, v_n$  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  $v_1, \dots, v_n$  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.

# **COMP3121/9101**

## **ALGORITHM DESIGN**

### **PRACTICE PROBLEM SET 1 – DATA STRUCTURES REVISION**

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

## **Contents**

|  |           |
|--|-----------|
| <b>1 SECTION ONE: DATA STRUCTURES AND ALGORITHMS</b>   | <b>2</b>  |
| <b>2 SECTION TWO: SEARCHING AND SORTING ALGORITHMS</b> | <b>10</b> |
| <b>3 SECTION THREE: TIME COMPLEXITY ANALYSIS</b>       | <b>16</b> |

## § SECTION ONE: DATA STRUCTURES AND ALGORITHMS

**[K] Exercise 1.** Let  $A$  be an array with  $n - 1$  elements, containing all integers from 1 to  $n$  except for one. Design an  $O(n)$  algorithm that finds the missing integer.

*Solution.* To design an algorithm that finds the missing integer, note that the sum of all integers from 1 to  $n$  is given by

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}.$$

Moreover, the missing integer (say  $x$ ) can be found by noting that

$$\begin{aligned} x &= 1 + 2 + \dots + (x-1) + x + (x+1) + \dots + n \\ &\quad - (1 + 2 + \dots + (x-1) + (x+1) + \dots + n) \\ &= \frac{n(n+1)}{2} - \sum_{i=1}^{n-1} A[i]. \end{aligned}$$

Therefore, our algorithm first computes  $n(n+1)/2$  in constant time and performs a running sum to compute  $\sum_{i=1}^{n-1} A[i]$  in linear time. The missing integer, therefore, is

$$x = \frac{n(n+1)}{2} - \sum_{i=1}^{n-1} A[i].$$

The overall algorithm runs in  $O(n)$  time since computing the sum of the elements in  $A$  takes linear time.  $\square$

**[K] Exercise 2.** Let  $A$  be an array with  $n$  elements. We say that  $A$  is *palindromic* if it can be read the same forwards and backwards. For example, the array  $A = [1, 2, 3, 2, 1]$  is palindromic while  $B = [1, 2, 3, 4, 2, 1]$  is not. Design an  $O(n)$  algorithm that determines whether  $A$  is a palindromic array.

*Solution.* To ensure that our array is *palindromic*, we need to ensure that  $A[i] = A[n-i+1]$  for each  $i = 1, \dots, n-1$ . Note that it is enough to check that the equality holds for  $i = 1, \dots, \lceil n/2 \rceil$  because any index  $i > \lceil n/2 \rceil$  is equivalent to an index from 1 to  $\lceil n/2 \rceil$  by symmetry, where  $\lceil x \rceil$  denotes the *smallest integer bigger than or equal to  $x$* .

Therefore, our strategy is as follows: we set up two pointers, one that keeps track of  $A[i]$  and one that keeps track of  $A[n-i+1]$ . If  $A[i] \neq A[n-i+1]$  for any  $i$  from 1 to  $\lceil n/2 \rceil$ , then we can terminate our algorithm and return that the array is not palindromic. Otherwise, if we have exhausted all possible indices and equality is satisfied, then we return that the array is palindromic.

The algorithm has a linear-time running time since we are looping through at most  $O(n/2) = O(n)$  times.  $\square$

**[K] Exercise 3.** Let  $A$  be an array with  $n$  distinct integers. You have to determine if there exist an integer (not necessarily in  $A$ ) which can be written as a sum of squares of two distinct integers from  $A$  in two different ways. Note that  $A[i]^2 + A[j]^2$  is treated the same as  $A[j]^2 + A[i]^2$ .

For example, if  $A = [1, 8, 9, 12]$ , then the integer 145 can be written as  $1^2 + 12^2$  and also  $8^2 + 9^2$ .

- (a) Design an  $O(n^2 \log n)$  algorithm that determines if such an integer exists in the *worst case*.
- (b) Design an algorithm that solves the same problem and runs in  $O(n^2)$  in the *expected case*.

*Solution.*

- (a) There are  $\binom{n}{2} = n(n - 1)/2$  pairs of indices  $1 \leq i < j \leq n$ . Now, for each such pairs, we compute the sum of the squares  $A[i]^2 + A[j]^2$  and store it in an array  $B$  of size  $n(n - 1)/2$ . Note that  $B$  now consists of all possible sums of squares of elements in  $A$ . The problem, therefore, reduces to the problem of determining whether there is a duplicate value in  $B$ . We can do this by first sorting  $B$  with merge sort, and then perform a linear scan for duplicate values; note that duplicate values must appear consecutively.

To argue the time complexity, we note that there are  $n(n - 1)/2 = O(n^2)$  many pairs of elements in  $A$ . Since  $B$  consists of  $O(n^2)$  many elements, sorting  $B$  takes  $O(n^2 \log(n^2))$ . But, by our log properties,  $n^2 \log(n^2) = 2n^2 \log n = O(n^2 \log n)$ . Therefore, sorting  $B$  takes  $O(n^2 \log n)$  running time. Finally, the linear scan takes  $O(n^2)$  running time which implies that the overall running time of our algorithm is  $O(n^2 \log n)$ .

- (b) As above, compute each sum  $A[k]^2 + A[m]^2$ , but instead store the sums in a hash table. Before adding a sum to the hash table, we check whether the same sum already appears there, if so, report **yes**.

For each of  $O(n^2)$  pairs of indices, we perform expected  $O(1)$  work, so the overall time complexity is expected  $O(n^2)$ .

□

**[K] Exercise 4.** Let  $A$  be an array with  $n$  integers, and let  $k$  be a positive integer. You have to determine if there exist two integers in  $A$  whose absolute difference is exactly  $k$ . In other words, you want to determine if there exist distinct indices  $i, j$  such that  $|A[i] - A[j]| = k$ .

- (a) Design an  $O(n \log n)$  algorithm that determines if two such integers in  $A$  exist.
- (b) Design an algorithm that solves the same problem and runs in  $O(n)$  in the *expected case*.

*Solution.*

- (a) The idea is that we can check for a pair of elements whose difference of  $k$  first, and then independently check for a pair of elements whose difference is  $-k$ . To do this efficiently, start by sorting  $A$  using merge sort which gives  $O(n \log n)$  time for sorting. Then, with two pointers, have one pointer for the bigger element and one pointer for the smaller element. Let  $i$  be the index of the larger value and  $j$  be the index of the smaller value.

Given the two indices, check for their difference and see if the difference is equal to  $k$ . If it is equal to  $k$ , then there is nothing to do. Otherwise, either  $A[i] - A[j] > k$  or  $A[i] - A[j] < k$ . We consider each case separately.

- **Case 1:**  $A[i] - A[j] > k$ . In this case, we observe that any value larger than  $A[j]$  will also not work. To see this, suppose that  $A[j] < A[j']$ . Then

$$A[i] - A[j'] > A[i] - A[j] > k.$$

Therefore, we only need to consider all indices  $j' < j$ .

- **Case 2:**  $A[i] - A[j] < k$ . In this case, we observe that any value smaller than  $A[i]$  will also not work. To see this, suppose that  $A[i] > A[i']$ . Then

$$A[i'] - A[j] < A[i] - A[j] < k.$$

Therefore, we only need to consider all indices  $i' > i$ , which reduces the search space for  $i$  under this constraint.

With these observations in mind, we now design our general algorithm. From sorting  $A$  in  $O(n \log n)$  with merge sort, we now consider two indices  $j, i$  that points to the first two elements in the array respectively.

- If  $A[i] - A[j] = k$ , then we terminate and return that there are such pairs in  $A$ .

- If  $A[i] - A[j] > k$ , then we are in case 1 and by our observation above, we decrease the pointer for  $j$  to consider indices smaller than  $j$ .
- If  $A[i] - A[j] < k$ , then we are in case 2 and by our observation above, we increase the pointer for  $i$  to consider indices larger than  $i$ .

If we exhaust all possible indices without terminating early, then there is no such pair whose difference is  $k$ . We repeat this same process except replace  $k$  with  $-k$ . If we exhaust all possible indices without terminating early with difference  $-k$  as well, then there is no pair of points whose absolute difference is  $k$ , in which case we return without a match.

Note that both parts of the algorithm perform exactly the same operations. Since we are doing a linear scan through the entire array once, checking for a match takes running time  $O(n)$  in the worst case, which is dominated by the running time of the running time of merge sort. Since we are doing this algorithm twice, the overall running time is  $O(2n \log n) = O(n \log n)$ .

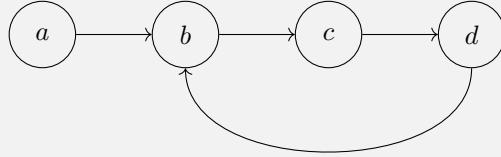
- (b) We create a hash table  $H$ , which is initially empty. Now, for each element  $a$  in  $A$ , we first perform a check to see whether  $a$  is already hashed in  $H$ . If it hasn't been hashed before, then we hash  $a + k$  and  $a - k$  into  $H$ . Otherwise, we must have hashed a previous element from  $A$  into  $a$ . But by the way that we have hashed our values, this implies that  $b + k = a$  or  $b - k = a$  for some element  $b$ . But this implies that either  $a - b = k$  or  $b - a = k$ , in which case we can terminate and return that there is such a pair of elements in  $A$ .

This operation is  $O(n)$  in the *expected* time because each time we search for an element in the hash table, the operation is expected  $O(1)$  time complexity. Doing this for all  $n$  elements in  $A$  gives us  $O(n)$  expected time complexity.

□

**[K] Exercise 5.** Let  $L$  be a linked list with  $n$  nodes. We say that  $L$  is *cyclic* if there is a node that can be reached again by continuously following the next node.

For example, the linked list visualised as



is cyclic because node  $b$  can be reached again from  $d$ . Design an  $O(n)$  algorithm that determines whether a linked list is cyclic.

*Solution.* The solution is useful to know since it introduces an algorithm called *Floyd's Cycle Detection algorithm*, also colloquially named the *tortoise and hare algorithm*, which can be used any time a problem asks to detect whether a cycle is present.

We first present the algorithm in its entirety and then prove why it works. The *tortoise and hare algorithm* is a prototypical two pointer algorithm, where one pointer moves in one step and the other pointer moves in two steps. Therefore, the algorithm is as follows: we initialise the tortoise and hare pointers to the head. Now, while the hare pointer isn't at the end of the linked list, we iterate through the entire list in two different speeds:

At every iteration,

- Move the tortoise pointer by one space (i.e.  $i \mapsto i + 1$ );
- Move the hare pointer by two spaces (i.e.  $i \mapsto i + 2$ ).

If the two pointers ever meet in the same place, then there must have been a cycle in the linked list and our algorithm returns that the linked list is cyclic. Otherwise, if we reach the end of the linked list and the two pointers never met, then there must have been no cycle in the linked list.

To see why this algorithm is correct, we need to prove two things; the first is to prove that, *if* a linked list contains a cycle, then the hare and tortoise pointers will eventually meet. The second is to prove that, *if* a linked list does not contain a cycle, then the hare and tortoise pointers will never meet. We prove these separately.

- Suppose that the linked list does contain a cycle. Denote the length of the cycle as  $C$  and suppose that the number of nodes before arriving at the start of a cycle is  $T$ . Firstly, we can write  $T = kC + r$  for some integer  $k$  and some  $0 \leq r < C$ . This allows us to track where the hare's pointer is when the tortoise pointer finally reaches the beginning of the cycle. When the tortoise is at the beginning of the cycle (at node  $T$ ), the hare will be at node  $2T = 2(kC + r)$ . However, the first  $T$  steps will be in the nodes leading up to the cycle. Therefore, in the cycle, the hare will have moved  $T$  steps. This implies that the hare will have moved around the cycle  $k$  times, leaving the hare on node  $r$  inside the cycle.

We now have a measure of where the tortoise and hare are in the cycle. We now observe that, after  $C - r$  more steps, the tortoise will be on node  $C - r$  in the cycle (think of entering the cycle as node 0 now). Since the hare was  $r$  moves ahead of the tortoise when the tortoise is inside the cycle, the hare must be on node  $r + 2(C - r)$  since the hare was an  $r$  number of steps ahead of the tortoise and then we had  $C - r$  many more iterations of the algorithm. But this implies that the hare is on node  $r + 2C - 2r = 2C - r$ . This is equivalent to stepping around the cycle once and ending back at node  $C - r$ . But this implies that the hare and the tortoise are both on node  $C - r$  in the cycle, which implies that both pointers met at node  $C - r$ . This completes the proof that a cycle inside a linked list leads to both pointers meeting at some point inside the cycle.

- Now suppose that the linked list does not contain a cycle. Then the linked list consists of paths, but this implies that the index of the hare will never be any smaller than the index of the tortoise. This shows that the hare will never meet the tortoise, which completes the proof.

This shows that the hare and tortoise will meet at the same node if and only if the linked list contains a cycle. This proves the correctness of the algorithm.  $\square$

**[H] Exercise 6.** You are at a party attended by  $n$  people (not including yourself), and you suspect that there might be a celebrity present. A *celebrity* is someone known by everyone, but who does not know anyone else present. Your task is to work out if there is a celebrity present, and if so, which of the  $n$  people present is a celebrity. To do so, you can ask a person  $X$  if they know another person  $Y$  (where you choose  $X$  and  $Y$  when asking the question).

- Show that there can be at most one celebrity. In other words, *if* a celebrity exists, then the celebrity is unique.
- Use the previous part to show that your task can always be accomplished by asking no more than  $3n - 3$  such questions.
- Show that your task can always be accomplished by asking no more than  $3n - \lfloor \log_2 n \rfloor - 3$  such questions.

*Solution.* Assume that the people are assigned a number from 1 to  $n$ .

- We make the following observation.

**Observation.** *There can be at most one celebrity.*

Suppose that there are two celebrities, say  $A$  and  $B$ . Since  $A$  is a celebrity,  $B$  must know  $A$  since everyone knows the celebrity. But then this implies that  $A$  cannot be a celebrity since a celebrity does not know anyone. So there can only be *at most* one celebrity. With this observation, we proceed as follows.

Arbitrarily pick any two people, say  $A$  and  $B$ . Ask if  $A$  knows  $B$ .

- If  $A$  knows  $B$ , then we know that  $A$  cannot be a celebrity.

- If  $A$  does not know  $B$ , then we know that  $B$  cannot be a celebrity.

In either case, we can eliminate one person from our *celebrity candidate* list. Since there are  $n$  people, we can ask  $n - 1$  people to find the *celebrity candidate*. Let this candidate be  $C$ .

We now check if  $C$  is indeed the celebrity (it could very well be the case that  $C$  is not the celebrity and as such, no celebrity exists). For each person  $X$  in the party, we ask the following question:

Does  $X$  know  $C$ ?

- If  $X$  does not know  $C$ , then we conclude that  $C$  cannot be the celebrity and thus, no celebrity is present.
- Otherwise,  $C$  may still be the celebrity.

We finally need to check if  $C$  knows anyone in the party. Again, choosing every person  $X$  in the party, we ask the following question:

Does  $C$  know  $X$ ?

- If  $C$  knows  $X$ , then we conclude that  $C$  cannot be the celebrity and thus, no celebrity is present.
- If  $C$  does not know  $X$ , we continue until we have exhausted everyone in the party.

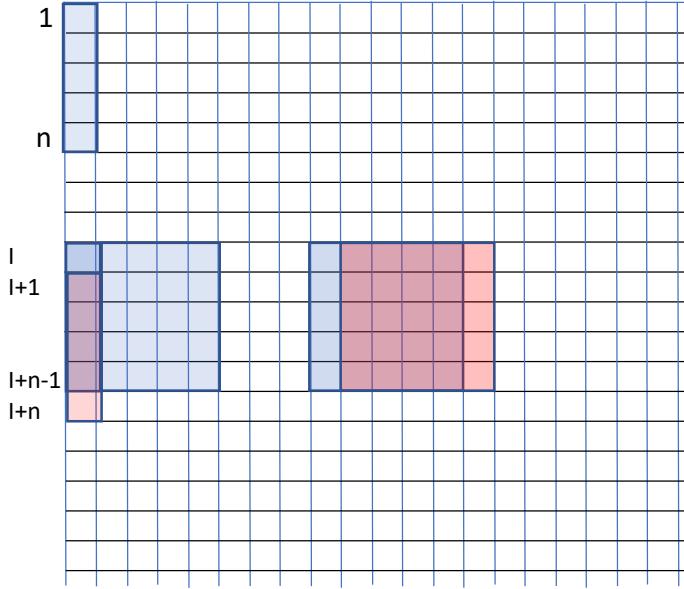
We only conclude that  $C$  must be the only celebrity once we have concluded that  $C$  does not know anyone. In the worst case, we consume  $n - 1$  questions to determine who the potential celebrity candidate is and then  $2(n - 1)$  questions to verify whether  $C$  is indeed the celebrity. Thus, in the worst case, we use  $(n - 1) + 2(n - 1) = 3n - 3$  questions in total.

- (b) We arrange  $n$  people present as leaves of a **balanced full tree**, i.e., a tree in which every node has either 2 or 0 children and the depth of the tree is as small as possible. To do that compute  $m = \lfloor \log_2 n \rfloor$  and construct a perfect binary tree with  $2^m \leq n$  leaves. If  $2^m < n$  add two children to each of the leftmost  $n - 2^m$  leaves of such a perfect binary tree. In this way you obtain  $2(n - 2^m) + (2^m - (n - 2^m)) = 2n - 2^{m+1} + 2^m - n + 2^m = n$  leaves exactly, but each leaf now has its pair, and the depth of each leaf is  $\lfloor \log_2 n \rfloor$  or  $\lfloor \log_2 n \rfloor + 1$ . For each pair we ask if, say, the left child knows the right child and, depending on the answer as in (a) we promote the potential celebrity one level closer to the root. It will again take  $n - 1$  questions to determine a potential celebrity, but during the verification step we can save  $\lfloor \log_2 n \rfloor$  questions (one on each level) because we can reuse answers obtained along the path that the potential celebrity traversed through the tree. Thus,  $3n - 3 - \lfloor \log_2 n \rfloor$  questions suffice.

□

**[H] Exercise 7.** You are in a square orchard of  $4n$  by  $4n$  equally spaced trees. You want to purchase apples from precisely  $n^2$  many of those trees, which also form a square. Fortunately, the owner is allowing you to choose such a square anywhere in the orchard and you have a map with the number of apples on each tree. Your task is to choose a square that contains the largest amount of apples and which runs in time  $O(n^2)$ .

*Solution.* We start by noting that there is a heavy overlap between such possible squares and we should use this fact to compute the number of apples in all of such squares in an efficient way. Consider to the figure below.



**Setup:** Let us visualize the orchard as a  $4n \times 4n$  discrete grid and let  $C(i, j)$  denote the cell at the coordinate  $(i, j)$ . Also let  $A[i, j]$  denote the amount of apples at  $C[i, j]$ .

**Step 1:** Now, we start by examining the first column by computing the sum  $\alpha(1, 1) = \sum_{k=1}^n A[k, 1]$ , (corresponding to cells  $C[1, 1]$  to  $C[n, 1]$  shown in blue in the top left corner of the orchard map) which takes  $n - 1 = O(n)$  additions. We then compute the number of apples  $\alpha(i, 1)$  in all rectangles  $r(i, 1)$  consisting of cells  $C[i, 1]$  to  $C[i + n - 1, 1]$  for all  $i$  such that  $2 \leq i \leq 3n + 1$ , starting from  $\alpha(1, 1)$  and using recurrence

$$\alpha(i + 1, 1) = \alpha(i, 1) - A[i, 1] + A[i + n, 1].$$

We know that the recurrence is valid as the rectangle (denoted with  $r(i, 1)$ ) consisting of cells  $C[i, 1]$  to  $C[i + n - 1, 1]$  and rectangle  $r(i + 1, 1)$  consisting of cells  $C[i + 1, 1]$  to  $C[i + n, 1]$  in the first column overlap and differ only in the first square of  $r(i)$  and the last square of  $r(i + 1)$  (this idea is similar to Question 1 but embedded 2 dimensions). Since each recursion step involves only one addition and one subtraction, this can all be done in  $O(n)$  steps.

**Step 2:** We can now apply the same procedure to each different column  $j$ , thus obtaining the number of apples  $\alpha(i, j)$  in every rectangle consisting of cells  $C(i, j)$  to  $C(i + n - 1, j)$ . As each column requires  $O(n)$  many computations, it takes  $O(n^2)$  many computations in total.

**Step 3:** Now for  $1 \leq i \leq 3n + 1$ , we compute  $\beta(i, 1) = \sum_{p=1}^n \alpha(i, p)$ . This gives the total number of apples acquired in the square with corner vertices at cells  $C(i, 1), C(i, n), C(i + n - 1, 1)$  and  $C(i + n - 1, n)$  (square with vertices along a single column). For  $n$  such computations it takes  $O(n)$  additions each, thus the total number of additions then runs in  $O(n^2)$ .

**Step 4:** So for each fixed  $i$  such that  $1 \leq i \leq 3n + 1$ , with the value of  $\beta(i, 1)$ , we can now compute  $\beta(i, j)$  for all  $2 \leq j \leq 3n + 1$  using recursion

$$\beta(i, j + 1) = \beta(i, j) - \alpha(i, j) + \alpha(i, j + n).$$

because the two adjacent squares overlap, except for the first column of the first square and the last column of the second square (overlap between red and blue square), the rest of the considered squares overlap. Each step of recursion takes only one addition and one subtraction so the whole recursion takes  $O(n)$  many steps. Thus, recursions for all  $1 \leq i \leq 3n + 1$  takes  $O(n^2)$  many operations. (idea is again similar to Question 1 but in 2 dimensions).

**Step 5:** Lastly, we only require to find the largest value of  $\beta(i, j)$  among all  $1 \leq i, j \leq 3n + 1$ , giving a complexity of  $O((3n)^2) = O(n^2)$  for finding the maximum.

As all required steps of the algorithm runs in a complexity of  $O(n^2)$ , the total complexity of the algorithm is then  $O(n^2)$  as required.  $\square$

**[E] Exercise 8.** There are  $n$  teams in the local cricket competition and you happen to have  $n$  friends that keenly follow it. Each friend supports some subset (possibly all or none too) of the  $n$  teams. Not being the sporty type, but wanting to fit in nonetheless, you must decide for yourself which subset of teams (again, possibly all or none too) to support.

You don't want to be branded as a copy cat so your subset must not be identical to anyone else's. The trouble is, you don't know which friends support which teams but you can ask your friends some question of the form: "does friend  $A$  support team  $B$ ?" (you choose  $A$  and  $B$  in advance).

Design a strategy that determines a suitable subset of teams for you to support and asks the fewest number of questions as possible.

Given your  $n$  friends, how many questions do you have to at least ask each friend? Can we generate a strategy just asking these questions?

*Solution.* Suppose your friends are numbered 1 to  $n$  and the teams are also numbered 1 to  $n$ . Then, for each  $i$ , ask friend  $i$  if they support team  $i$ . If they do, we choose not to support them and if they don't, we do support them. To see why this is different to all of our friends, if it were the same as *some* friend, then, in each index, our lists must agree. However, when we ask friend  $i$  whether they support team  $i$ , our algorithm forces us to choose a different answer to theirs, which means it cannot possibly agree for every team. Therefore, our subset of teams that we support cannot possibly align with all of the choices that any other friend makes. This also uses  $n$  queries which is minimal since we require some information about *every* possible friend.  $\square$

**[E] Exercise 9.** You are conducting an election among a class of  $n$  students. Each student casts precisely one vote by writing their name and that of their chosen classmate on a single piece of paper. However, the students have forgotten to specify the order of names on each piece of paper; for example, "Alice Bob" could mean that *Alice voted for Bob* or *Bob voted for Alice*.

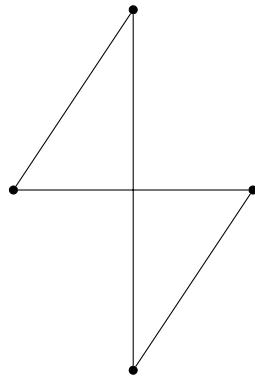
- (a) Show how you can still uniquely determine how many votes each student received.
- (b) Hence, explain how you can determine which students did not receive any votes. Can you also determine who these people voted for?
- (c) Suppose now that every student received at least one vote. Show that each student received *exactly* one vote.
- (d) Using the previous two parts, design an  $O(n)$  algorithm that constructs a list of votes of the form " $X$  voted for  $Y$ " consistent with the pieces of paper. Specifically, each piece of paper should match up with precisely one of these votes. If multiple such lists exist, produce any.

First, use (c) to consider how you would solve it in the case where every student received at least one vote. Then apply (b).

*Solution.* We make the following observation.

**Observation.** Every student has their name on *at least* one paper. To see this, note that every student casts a vote by writing two names. One name signifies that they are the voter and the other being who they voted for. Therefore, because every student votes, their name must appear on *at least* one paper.

- (a) To see how we can uniquely determine how many votes each student receives, we make use of the assumption that every student casted precisely one vote. Therefore, if their name appeared on  $x$  many papers, precisely one of these papers must have been the paper that they used to vote on. In other words, they have received  $x - 1$  many votes.
- (b) Using the previous part, if they received 0 votes and the observation, we deduce that they must have appeared on precisely 1 paper (the paper that they voted on!). The person that they voted is precisely the other name on the single paper.
- (c) If every student received at least one vote, then we require *at least*  $n$  distinct pieces of papers which correspond to the distinct votes. However, there are *at most*  $n$  papers because there are maximally  $n$  students, each of whom cast a vote. Therefore, every student can receive *at most* 1 vote. In other words, if every student received at least one vote, then they receive *exactly* 1 vote.
- (d) We begin with the case explored in part (c). Suppose that every student received at least one vote. Then we deduced that they received *exactly* 1 vote. We can represent the following scenario as an undirected graph, where each of the vertices represent a student and each edge represents a vote. That is,  $x$  and  $y$  are connected if either  $x$  voted for  $y$  or  $y$  voted for  $x$ .



Above is an example of such a graph. One key property is that each student must appear on *exactly* two papers; these correspond to the edges of any particular vertex and correspondingly, it is easy to see that any such graph can be partitioned into a disjoint union of cycle subgraphs.

Pick any student  $s$  appearing on two pieces of paper, and arbitrarily choose one of their pieces of paper as their vote. Suppose they voted for  $t$ . We are now left with a single choice for  $t$ 's vote. We can repeatedly follow these pieces of paper until we arrive back to  $s$ . We then repeat with another student appearing on two pieces of paper until all votes have been resolved. We can do this in  $O(n)$  altogether, for instance, using a (simplified) Depth-First Search (DFS).

Now we combine this with part (b) to obtain an algorithm for the general case. We repeatedly check if a student has no votes (by counting votes) and resolve their vote. Once we reach a point where this is no longer possible, we know every student received at least one vote, and use the algorithm above.

This can be done in  $O(n^2)$  by repeatedly taking  $O(n)$  to identify a student who has no votes, or more cleverly in  $O(n)$  as follows. We keep a count, for each student, how many pieces of paper they appear on and maintain a queue of students who appear on only one piece of paper. We can initially populate this queue in  $O(n)$ . Then, we repeatedly process the front student of the queue by removing their vote. Note that this *only changes the vote count of the person they voted for*, so we simply decrease their count. If their count reaches 1, we push them onto the queue. Hence, we process each student, updating counts and the queue in  $O(1)$  so this step is  $O(n)$  as well, giving an  $O(n)$  algorithm.

□

## § SECTION TWO: SEARCHING AND SORTING ALGORITHMS

### [K] Exercise 10.

- Assume you have an array of  $2n$  distinct integers. Find the largest and the smallest number using at most  $3n - 2$  comparisons.
- Assume you have an array of  $2^n$  distinct integers. Find the largest and second largest number using at most  $2^n + n - 2$  comparisons.

*Solution.*

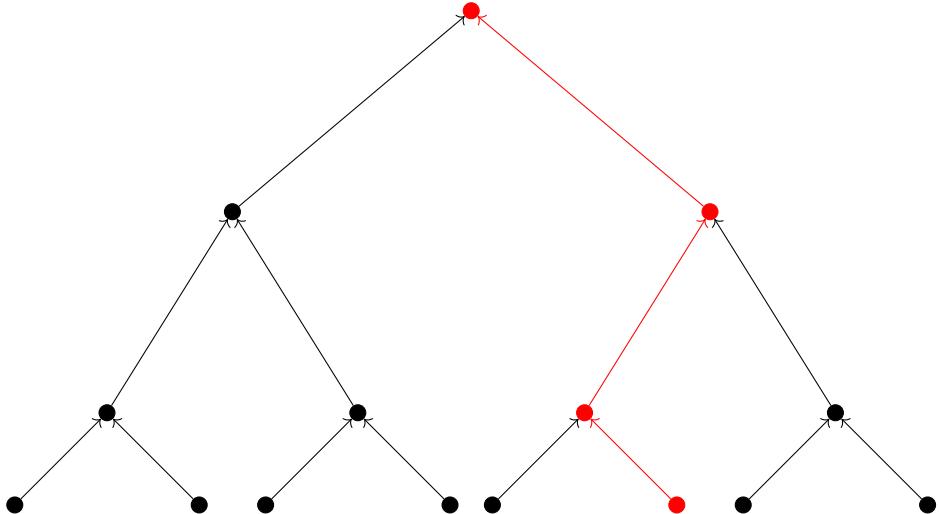
- Consider first the brute force algorithm:

- Compare the first two elements  $A[1]$  and  $A[2]$ , and set  $m$  as the smaller of them and  $M$  as the larger.
- For each of the following  $2n - 2$  elements, compare it with both  $m$  and  $M$ , updating either if necessary.

In the best case, we will update  $m$  at each step, making it unnecessary to ever compare against  $M$ , and resulting in a total of  $2n - 1$  comparisons. However, this algorithm is not acceptable because in the worst case, each of  $2n - 2$  elements requires two comparisons, for a total of  $2(2n - 2) + 1 = 4n - 3$  comparisons.

Instead, we first form  $n$  pairs and compare the two elements of each pair, putting the smaller into a new array  $S$  and the larger into a new array  $L$ . Note that the smallest of all  $2n$  elements must be in  $S$  and the largest in  $L$ , and we have made  $n$  comparisons. We now use two linear searches for the minimum of  $S$  and the maximum of  $L$ , each taking  $n - 1$  comparisons. In total this takes  $n + n - 1 + n - 1 = 3n - 2$  comparisons.

- Consider the figure below.



We see a complete binary tree with  $2^n$  leaves and  $2^n - 1$  internal nodes and of depth  $n$  (the root has depth 0). We then place all the numbers at the leaves, compare each pair and “promote” the larger element (shown in red) to the next level and proceed in such a way till you reach the root of the tree, which will contain the largest element.

Clearly, each internal node is a result of one comparison and there are  $2^n - 1$  many nodes thus also the same number of comparisons so far. Now just note that the second largest element must be among the black nodes which were compared with the largest element along the way - all elements underneath them must be smaller or equal to the elements shown in black. There are  $n$  many such elements so finding the largest among them will take  $n - 1$  comparisons by brute force.

In total, requires at most  $2^n + n - 2$  many comparisons.

□

**[K] Exercise 11.** You are given an array  $A$  of  $n$  integers and an integer  $x$ .

- (a) Design an  $O(n \log n)$  algorithm that determines whether or not there exist two integers in  $A$  that sum to  $x$ .
- (b) Design an algorithm that solves the same problem and runs in  $O(n)$  **expected** time.

*Solution.*

(a) Note that a brute force solution considers all possible pairs of  $(A[i], A[j])$  does not suffice as it runs in  $O(n^2)$  time. Instead, we start by sorting the array in ascending order, which can be done in  $O(n \log n)$  in the worst case using an algorithm such as MERGE SORT. Here we give 2 valid approaches.

**Approach 1:** For each element  $a$  in the array, we can check if there exists an element  $x - a$  also in the array in  $O(\log n)$  time using binary search. The only special case is if  $a = x - a$  (i.e.  $x = 2a$ ), where we just need to check the two elements adjacent to  $a$  in the sorted array to see if another  $a$  exists.

Hence, we take at most  $O(\log n)$  time for each element, so this part is also  $O(n \log n)$  time in the worst case, giving an  $O(n \log n)$  algorithm.

**Approach 2:** Alternatively, we add the smallest and the largest elements of the array. If the sum exceeds  $x$  no solution can exist involving the largest element; if the sum is smaller than  $x$  then no solution can exist involving the smallest element. Thus, if this sum is not equal to  $x$  we can eliminate 1 element.

After at most  $n - 1$  many such steps you will either find a solution or will eliminate all elements except one, thus verifying no such elements exist. This takes  $O(n)$  time in the worst case, so the overall running time is  $O(n \log n)$  as the sorting dominates.

- (b) We take a similar approach in part (a), except we use a hash map (or hash table, denoted  $H$ ) to check if elements exist in the array (rather than sorting it): each insertion and lookup takes  $O(1)$  **expected** time. We again provide 2 valid approaches.

**Approach 1:** At index  $i \in \{1, 2, \dots, n\}$ , we assume the previous  $i - 1$  elements of  $A$  are already stored in  $H$ . Then we check if  $x - A[i]$  is in  $H$  in expected  $O(1)$ , then insert  $A[i]$  into  $H$ , also in expected  $O(1)$ .

As this process will take  $n$  insertions and  $n$  look-ups in the worst case, we conclude that the algorithm runs in  $O(2n) = O(n)$  **expected** time.

**Approach 2:** Alternatively, we hash all elements of  $A$  and store its occurrence frequency. We then go through elements of  $A$  again, this time for each element  $a$  we check if  $x - a$  is in  $H$ . However, if  $2a = x$ , we must also check if at least 2 copies of  $a$  appear in the corresponding slot  $H$ .

As this process will take again  $n$  insertions and  $n$  look-ups in the worst case, hence the algorithm runs in  $O(n)$  **expected** time.

□

**[K] Exercise 12.** Let  $f : \mathbb{N} \rightarrow \mathbb{Z}$  be a monotonically increasing function. That is, for all  $i \in \mathbb{N}$ , we have that  $f(i) < f(i + 1)$ . Our goal is to find the smallest value of  $i \in \mathbb{N}$  so that  $f(i) \geq 0$ . Design an  $O(\log n)$  algorithm to find the value of  $i$  so that  $f(i) \geq 0$  for the first time.

How could you use binary search here?

*Solution.* This is similar to the problem 2 of tutorial 1.

If we wanted to apply binary search here, we need some sort of range of values to look through; however, we don't have a suitable range to binary search over. We can, instead, construct a more dynamic range of values. To see this in action, we can start by computing the first two values. Computing  $f(1)$  and  $f(2)$  gives us two values to work with. If  $f(1) < 0 \leq f(2)$ , then we are done. Similarly, if  $0 < f(1) < f(2)$ , then we are also done. Therefore, we may assume that  $f(1) < f(2) < 0$ . For any subsequent iteration, we compute the function at the next power of two, say  $2^k$ . If  $f(2^k) < 0$ , then we compute  $f(2^{k+1})$ . However, if  $f(2^k) \geq 0$ , then we know that the first index in which the function is non-negative must lie in the range  $[2^{k-1}, \dots, 2^k]$ . Since the function is monotonically increasing, this allows us to perform a binary search.

Since we're computing the function at powers of two, the number of times we are performing our computation is  $\log_2 n$ , where  $n = 2^k$  since we are performing this over  $k$  iterations. Now, since we're binary searching over the interval  $[2^{k-1}, \dots, 2^k]$ , the length of our array is  $2^k - 2^{k-1} = 2^{k-1}$ . Therefore, the running time of our binary search is  $\log(2^{k-1}) = k - 1 < k = \log n$ . Therefore, the overall running time of our algorithm is  $O(\log n)$ .  $\square$

**[K] Exercise 13.** Let  $M$  be an  $n \times n$  matrix of distinct integers  $M(i, j)$  where  $1 \leq i, j \leq n$ . Each row and each column of the matrix is sorted in increasing order, so that for each row  $i$ ,

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

and for each column  $j$ ,

$$M(1, j) < M(2, j) < \dots < M(n, j).$$

Design an  $O(n)$  algorithm that, given an integer  $x$ , determines whether  $M$  contains the integer  $x$ .

*Solution.* Consider  $M(1, n)$  (i.e., top right cell).

- If  $M(1, n) = x$ , we are done.
- If  $M(1, n) < x$ , the number  $x$  is not found in the top row because  $M(1, 1) < M(1, 2) < \dots < M(1, n) < x$ . We can therefore ignore this row.
- If  $M(1, n) > x$ , then similarly  $x$  cannot be found in the rightmost column because all other elements there are larger than  $M(1, n)$ . Thus the last column can be ignored.

In the worst case, the sum of the width and height of the search table is reduced by one. We continue in this manner until either  $x$  is found or we reach an empty table and thus ascertain that  $x$  does not occur in the table. Since the initial sum of the height and the width of the table is  $2n$  and at each step we make only one comparison, the algorithm takes  $O(n)$  many steps.  $\square$

**[H] Exercise 14.** You are given two arrays,  $A$  and  $B$ , each containing  $n$  distinct positive integers each. Let  $f(x, y) = y^6 + x^4y^4 + x^2y^2 - x^8 + 10$ .

- For any fixed value  $x$ , show that  $f(x, y)$  is an increasing function in terms of  $y$ .
- Hence, design an  $O(n \log n)$  algorithm that determines if  $A$  contains a value for  $x$  and  $B$  contains a value for  $y$  such that  $f(x, y) = 0$ .

*Solution.*

- Here, we make a simplifying assumption that  $x, y$  are both positive integers. Fix a value of  $x$ . Then  $f_y(x, y) = 6y^5 + 4x^4y^3 + 2x^2y > 0$  whenever  $y > 0$ . Therefore, for positive inputs of  $y$ , we see that  $f_y(x, y) > 0$  which shows that it is increasing in the input of  $y$ .

- (b) From the above observation, we can perform a binary search by sorting  $B$  using merge sort and binary search for values of  $y$ . For each  $x \in A$ , binary search for a value of  $y$  such that  $f(x, y) = 0$ . If  $f(x, y) > 0$ , then we recurse on the top half of the array  $B$ ; otherwise, we recurse on the bottom half of  $B$ . If we have exhausted all possible values of  $B$ , then there are no possible pairs  $(x, y)$  that satisfy the equation. This is  $O(n \log n)$  since merge sort is  $O(n \log n)$  in the worst case and we perform an  $O(\log n)$  operation on  $n$  many values, giving us a final complexity of  $O(n \log n)$ .

□

**[H] Exercise 15.** Let  $A$  be an array with  $n$  integers. You need to answer a series of  $n$  queries, each of which is of the form “*how many elements a of the array A satisfy  $L_k \leq a \leq R_k$ ?*”, where  $L_k, R_k$  (for some  $1 \leq k \leq n$ ) are integers such that  $L_k \leq R_k$ . Design an  $O(n \log n)$  algorithm that answers each of these  $n$  queries.

*Solution.* We start by sorting  $A$  in  $O(n \log n)$  in ascending order, using MERGE SORT. Then, for each query, we can 2 binary searches to find the indexes (denote with  $(i, j)$ ) of the:

- First element with value **no less** than  $L$ ; and
- First element with value **strictly greater** than  $R$ .

The difference between these indices is the answer to the query, i.e.,  $j - i$ . Note that if your binary search hits  $L$  you have to see if the preceding element is smaller than  $L$ ; if it is also equal to  $L$ , you have to continue the binary search (going towards the smaller elements) until you find the first element equal to  $L$ . A similar observation applies if your binary search hits  $R$ .

Each binary search then takes  $O(\log n)$  so for  $n$  queries in total, the algorithm runs in  $O(n \log n)$  overall. □

**[H] Exercise 16.** Suppose that you are taking care of  $n$  kids, who took their shoes off. You have to take the kids out and it is your task to make sure that each kid is wearing a pair of shoes of the right size (not necessarily their own, but one of the same size). All you can do is to try to put a pair of shoes on a kid, and see if they fit, or are too large or too small; you are NOT allowed to compare a shoe with another shoe or a foot with another foot. Describe an algorithm whose expected number of shoe trials is  $O(n \log n)$  which properly fits shoes on every kid.

*Solution.* This is done by a “double QUICKSORT” as follows. Pick a shoe and use it as a pivot to split the kids into three groups: those for whom the shoe was too large, those who fit the shoe and those for whom the shoe was too small. Then pick a kid for whom the shoe was a fit and let him try all the shoes, splitting them in three groups as well: shoes that are too small, shoes that fit him and the shoes which were too large for him. Continue this process with the first group of kids and first group of shoes and then also the third group of shoes with the third group of kids. If kids and shoes are picked randomly, the expected time complexity will be  $O(n \log n)$ . □

**[E] Exercise 17.** Your army consists of a line of  $n$  giants, each with a certain height. You must designate precisely  $\ell \leq n$  of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least  $k \geq 0$  giants standing in between them. Given  $n, \ell, k$  and the heights,  $H[1], H[2], \dots, H[n]$ , of the giants in the order that they stand in the line as input, find the *maximum* height of the *shortest* leader among all valid choices of  $\ell$  leaders. We call this the *optimisation* version of the problem.

For example, consider the following inputs:  $n = 10$ ,  $\ell = 3$ ,  $K = 2$ , and  $H = [1, 10, 4, 2, 3, 7, 12, 8, 7, 2]$ . Then, among the 10 giants, you must choose 3 leaders so that each pair of leaders has at least 2 giants standing in between them. The best choice of leaders has heights 10, 7 and 7, with the shortest leader having height 7. This is the best possible for this case.

- (a) In the *decision* version of this problem, we are given an additional integer  $T$  as input. Our task is to decide if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than  $T$ .

Give an algorithm that solves the decision version of this problem in  $O(n)$  time.

- (b) Hence, show that you can solve the optimisation version of this problem in  $O(n \log n)$  time.

*Solution.*

- (a) Notice that for the decision variant, we only care for each giant whether its height is at least  $T$ , or less than  $T$ : the actual value doesn't matter. Call a giant *eligible* if their height is at least  $T$ .

We sweep from left to right, taking the first eligible giant we can, then skipping the next  $K$  giants and repeating. We return `true` if the total number of giants we obtain from this process is at least  $L$ , or `false` otherwise. Each giant is processed in constant time, so the algorithm is clearly  $O(n)$ .

- (b) Observe that the optimisation problem corresponds to finding the largest value of  $T$  for which the answer to the decision problem is `true`.

Suppose our decision algorithm returns `true` for some  $T$ . Then clearly it will return true for all smaller values of  $T$  as well: since every giant that is eligible for this  $T$  will also be eligible for smaller  $T$ . Hence, we can say that our decision problem is *monotonic* in  $T$ .

Thus, we can use binary search to work out the maximum value of  $T$  where our decision problem returns `true`. Note that it suffices to check only heights of giants as candidate answers: the answer won't change between them. Thus, we can sort our heights in  $O(n \log n)$  and binary search over these values, deciding whether to go higher or lower based on a run of our decision problem. Since there are  $O(\log n)$  iterations in the binary search, each taking  $O(n)$  to resolve, our algorithm is  $O(n \log n)$  overall.

□

[E] **Exercise 18.** You are given  $n$  numbers  $x_1, \dots, x_n$ , where each  $x_i$  is a real number in the interval  $[0, 1]$ .

- (a) Describe an  $O(n \log n)$  algorithm that outputs a permutation  $y_1, \dots, y_n$  of the  $n$  numbers such that

$$\sum_{i=1}^n |y_i - y_{i-1}| < 2.$$

In other words, the sum of the absolute difference between adjacent elements is strictly smaller than 2.

- (b) Describe an  $O(n)$  algorithm that solves the same problem as the previous question.

Tweak the BUCKETSORT algorithm. What size buckets should we use?

*Solution.* We start by splitting the interval  $[0, 1]$  into  $n$  equal buckets, namely

$$B = \{b_0, b_1, \dots, b_{n-1}\} = \left\{ \left[0, \frac{1}{n}\right), \left[\frac{1}{n}, \frac{2}{n}\right), \dots, \left[\frac{n-1}{n}, 1\right) \right\}$$

Then we consider the function  $b : \mathbb{R} \rightarrow \{0, \dots, n-1\}$  which computes  $b(x) = k$  such that  $x \in b_k$ . Then we consider that the value  $x_i$  belongs to bucket number  $b(x_i) = \lfloor n x_i \rfloor$ .

*Proof.* From the definition, we know that  $x_i$  is in the bucket  $k$  if and only if

$$\frac{k}{n} \leq x_i < \frac{k+1}{n} \implies k \leq n x_i < k+1 \quad \text{then} \quad k = \lfloor n x_i \rfloor.$$

Therefore we can form  $n$  pairs  $\langle x_i, b(x_i) \rangle$ , each in constant time. Then we can now sort these pairs according to their bucket number  $b(x_i)$ ; since all bucket numbers are less than  $n$ , COUNTINGSORT does that in linear time. One can show that this sequence already satisfies the condition of the problem, but to make things simpler we do another extra step. We go through the sequence and in each bucket we find the smallest and the largest element; this can clearly be done in linear time.

We now slightly change the ordering of each bucket: we always start with the smallest element in that bucket and finish with the largest element (leaving all other elements in the same order). Discard the bucket numbers, leaving only a sequence  $y_j$  of real numbers between 0 and 1, which are the original  $x_i$  rearranged.

We now prove that this sequence satisfies  $\sum_{i=2}^n |y_i - y_{i-1}| < 2$ . We start by splitting this sum into two parts:

$$\begin{aligned} \sum_{i=2}^n |y_i - y_{i-1}| &= \sum_j (|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}) + \\ &\quad \sum_j (|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}). \end{aligned}$$

Note that there can be at most  $n - 1$  pairs of consecutive elements  $y_i, y_{i-1}$  which are in the same bucket (when all elements are in the same bucket). Whenever two elements  $y_i, y_{i-1}$  are in the same bucket,  $|y_i - y_{i-1}|$  is at most equal to the size of the bucket, i.e.,  $< \frac{1}{n}$ .

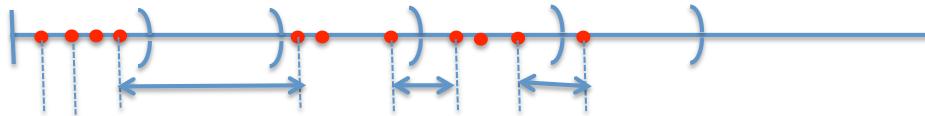
Thus,

$$\sum_j (|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in the same bucket}) < \frac{n-1}{n} < 1.$$

Also,

$$\sum_j (|y_j - y_{j-1}| : y_j \text{ and } y_{j-1} \text{ are in different buckets}) \leq 1$$

because each such pair joins the largest entry of one bucket to the smallest entry of the next non-empty bucket; as a result, the intervals in question are disjoint, as shown in the figure below.



Thus the total sum is smaller than  $1 + 1 = 2$ . □

## § SECTION THREE: TIME COMPLEXITY ANALYSIS

[K] **Exercise 19.** Show the following asymptotic relations by providing suitable constants for  $c$  and  $N$ .

- (a)  $n^2 = O(n^3)$ .
- (b)  $4n^4 + 8n^2 + 1 = \Omega(n^3)$
- (c)  $n^2 + \sin(n) = O(n^2)$ .
- (d)  $\frac{2n^2 + 3n + 1}{3n + 1} = \Theta(n)$ .
- (e)  $\cos(n) + \sin(n) = \Theta(1)$ .

*Solution.*

(a) Note that, for all  $n \geq 1$ ,  $n^2 \leq n \cdot n^2 = n^3$ . Therefore, choosing  $c = 1$  and  $N = 1$  shows that  $n^2 = O(n^3)$ .

(b) We want to find  $c, N > 0$  such that

$$c \cdot n^3 \leq 4n^4 + 8n^2 + 1,$$

for all  $n \geq N$ . If  $n > 8$ , then  $8n^2 < n^3 < n^4$  and  $1 < n^4$ . Therefore, we can bound the right hand side by  $4n^4 + n^4 + n^4 = 6n^4$  for all  $n > 8$ . Choosing  $c = 1$  and  $N = 8$  proves the result.

(c) Note that  $\sin(n) \leq 1$ . Therefore,  $n^2 + \sin(n) \leq n^2 + 1 \leq n^2 + n^2$  for all  $n > 1$ . Therefore, choose  $c = 2$  and  $N = 1$ .

(d) Using long division, we can show that

$$\frac{2n^2 + 3n + 1}{3n + 1} = \frac{2n}{3} + \frac{2}{9(3n + 1)} + \frac{7}{9}.$$

Since  $n > 0$ , we note that

$$f(n) \leq \frac{2n}{3}.$$

Therefore, let  $c = 2/3$  and  $N = 1$ .

(e) Note that  $\cos(n) + \sin(n) \leq \sqrt{2}$ . Therefore, let  $c = \sqrt{2}$  and  $N = 1$ .

□

[K] **Exercise 20.** Determine if  $f(n) = O(g(n))$ ,  $f(n) = \Omega(g(n))$ , both (i.e.  $f(n) = \Theta(g(n))$ ) or neither for the following pairs of functions. Justify your answer in each.

- (a)  $f(n) = (\log_2 n)^2$ ,  $g(n) = \log_2(n^{\log_2 n}) + 2\log_2 n$ .
- (b)  $f(n) = n^{100}$ ,  $g(n) = 2^{n/100}$ .
- (c)  $f(n) = \sqrt{n}$ ,  $g(n) = 2^{\sqrt{\log_2 n}}$ .
- (d)  $f(n) = n^{1.001}$ ,  $g(n) = n \log_2 n$ .
- (e)  $f(n) = n^{(1+\sin(\pi n/2))/2}$ ,  $g(n) = \sqrt{n}$ .

*Solution.*

(a) We see that

$$\begin{aligned} g(n) &= \log_2 n \cdot \log_2 n + 2 \log_2 n \\ &= \Theta((\log_2 n)^2) = \Theta(f(n)), \end{aligned}$$

since  $2 \log_2 n < (\log_2 n)^2$  for sufficiently large  $n$ .

(b) We show that  $f(n) = O(g(n))$ . To do this, we want to find some constants  $c, N > 0$  such that, for every  $n > N$ ,  $f(n) < c \cdot g(n)$ . Since  $\log$  is a monotonically increasing function,  $\log f(n) < \log g(n)$  immediately implies that  $f(n) < g(n)$ . We see that

$$\begin{aligned} \log_2 f(n) &= \log_2 (n^{100}) \\ &= 100 \log_2 n, \\ \log_2 g(n) &= \log_2 (2^{n/100}) \\ &= \frac{n}{100}. \end{aligned}$$

It therefore suffices to show that, for sufficiently large  $n$ ,  $10000 \log_2 n < n$ . We see that

$$\lim_{n \rightarrow \infty} \frac{10000 \log_2 n}{n} = \lim_{n \rightarrow \infty} \frac{10000}{n \log 2} = 0,$$

by L'Hôpital's rule. In other words, there exist some  $N > 1$  such that, for all  $n > N$ ,  $10000 \log_2 n/n < 1$  and thus,  $\log_2 f(n) < \log_2 g(n)$  which implies that  $f(n) < g(n)$  and so,  $f(n) = O(g(n))$ .

(c) We show that  $f(n) = \Omega(g(n))$ . This amounts to showing that  $\sqrt{n} > c \cdot 2^{\sqrt{\log_2 n}}$  for some  $c > 0$  and for all sufficiently large  $n$ . Since  $\log$  is monotonically increasing, this is equivalent to showing that

$$\log_2 \sqrt{n} = \frac{1}{2} \log_2 n > \log_2 c + \sqrt{\log_2 n} = \log_2 (c \cdot 2^{\sqrt{\log_2 n}}).$$

Taking  $c = 1$ , it is clear that  $\log_2 n$  grows asymptotically faster than  $\sqrt{\log_2 n}$ . Hence,  $\log_2 \sqrt{n} > \log_2 (2^{\sqrt{\log_2 n}})$  which implies that  $f(n) = \Omega(g(n))$ .

(d) We again wish to show that  $n^{1.001} = \Omega(n \log n)$ , i.e., that  $n^{1.001} > cn \log n$  for some  $c$  and all sufficiently large  $n$ . Since  $n > 0$  we can divide both sides by  $n$ , so we have to show that  $n^{0.001} > c \log n$ . We again take  $c = 1$  and show that  $n^{0.001} > \log n$  for all sufficiently large  $n$ , which is equivalent to showing that  $\log n/n^{0.001} < 1$  for sufficiently large  $n$ . To this end we use the L'Hôpital's to compute the limit

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log n}{n^{0.001}} &= \lim_{n \rightarrow \infty} \frac{(\log n)'}{(n^{0.001})'} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{0.001 n^{0.001-1}} \\ &= \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{0.001 \frac{1}{n} \cdot n^{0.001}} = \lim_{n \rightarrow \infty} \frac{1}{0.001 \cdot n^{0.001}} = 0. \end{aligned}$$

Since  $\lim_{n \rightarrow \infty} \frac{\log n}{n^{0.001}} = 0$  then, for sufficiently large  $n$  we will have  $\frac{\log n}{n^{0.001}} < 1$ .

(e) Just note that  $(1 + \sin \pi n/2)/2$  cycles, with one period equal to  $\{1/2, 1, 1/2, 0\}$ . Thus, for all  $n = 4k + 1$  we have  $(1 + \sin \pi n/2)/2 = 1$  and for all  $n = 4k + 3$  we have  $(1 + \sin \pi n/2)/2 = 0$ . Thus for any fixed constant  $c > 0$  for all  $n = 4k + 1$  eventually  $n^{(1+\sin \pi n/2)/2} = n > c\sqrt{n}$ , and for all  $n = 4k + 3$  we have  $n^{(1+\sin \pi n/2)/2} = n^0 = 1$  and so  $n^{(1+\sin \pi n/2)/2} = 1 < c\sqrt{n}$ . Thus, neither  $f(n) = O(g(n))$  nor  $f(n) = \Omega(g(n))$ .

□

**[K] Exercise 21.** Let  $f(n)$  and  $g(n)$  be two positive functions on  $\mathbb{N}$ . Prove that  $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$ .

*Solution.* Suppose that  $f(n) = O(g(n))$ . Then there exist constants  $c, n_0 > 0$  such that

$$f(n) \leq c \cdot g(n),$$

for all  $n \geq n_0$ . Then we have that

$$\frac{1}{c} f(n) \leq g(n),$$

for all  $n \geq n_0$ . Therefore, choose  $d = 1/c$ , which implies that  $g(n) = \Omega(f(n))$ . Now, suppose that  $g(n) = \Omega(f(n))$ . Then there exist constants  $c, n_0 > 0$  such that

$$c \cdot f(n) \leq g(n).$$

But again, this implies that

$$f(n) \leq \frac{1}{c} \cdot g(n).$$

Choosing  $d = 1/c$  again implies that  $f(n) = O(g(n))$ . Therefore,  $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$ .  $\square$

**[K] Exercise 22.** Let  $f(x) = x^2 - 12$ ,  $g(x) = x^3$ , and  $h(x) = 7x - 3$ . Show the following asymptotic relations.

- (a)  $f(x) = O(g(x))$ .
- (b)  $f(x) \cdot h(x) = \Theta(g(x))$ .
- (c)  $\frac{1}{f(x)} = O(1)$ .
- (d)  $\frac{1}{g(x)} = \Omega(e^{-x})$ .

*Solution.*

- (a) Observe that  $f(x) = x^2 - 12 < x^2 + 12 < x^2 + x$ , for all  $x > 12$ . But since  $x > 12$  is positive, this implies that

$$f(x) < x^2 + x^2 < \underbrace{x^2 + \cdots + x^2}_{x \text{ times}} = x \cdot x^2 = x^3 = g(x).$$

Therefore, for all  $x > 12$ ,  $f(x) < g(x)$ , which shows that  $f(x) = O(g(x))$ .

- (b) We now prove that  $f(x) \cdot h(x) = \Theta(g(x))$ . We first show that  $f(x) \cdot h(x) = O(g(x))$ . Consider

$$f(x) \cdot h(x) = 7x^3 - 3x^2 - 96x + 36.$$

For all  $x > 0$ , we note that

$$f(x) \cdot h(x) < 7x^3 + 3x^2 + 96x + 36.$$

For all  $x > 96$ , we see that  $3x^2 < 96x^2 < x^3$ ,  $96x < x^2 < x^3$ , and  $36 < x < x^3$ . Therefore,

$$f(x) \cdot h(x) < 7x^3 + x^3 + x^3 + x^3 = 10x^3.$$

Choosing  $c = 10$  and  $n_0 = 96$ , this shows that  $f(x) \cdot h(x) = O(g(x))$ . We now prove that  $f(x) \cdot h(x) = \Omega(g(x))$ . Clearly, we have that

$$f(x) \cdot h(x) > 7x^3 - 3x^2 - 96x.$$

Now, for all  $x > 96$ , we have that  $-x < -96$ . Therefore,  $f(x) \cdot h(x) > 7x^3 - 3x^2 - x^2 = 7x^3 - 4x^2$ . Similarly, since  $x > 4$ , we have that  $-x < -4$  which implies that  $-x^3 < -4x^2$ . But then this implies that

$$f(x) \cdot h(x) > 7x^3 - x^3 = 6x^3.$$

Therefore,  $f(x) \cdot h(x) > 6g(x)$  for all  $x > 96$ . Choosing  $c = 6$  and  $n_0 = 96$ , this shows that  $f(x) \cdot h(x) = \Omega(g(x))$  which shows that  $f(x) \cdot h(x) = \Theta(g(x))$ .

- (c) Note that  $1/f(x) \rightarrow 0$  as  $x \rightarrow \infty$ . Therefore, for large enough  $n_0$ , we can pick an arbitrary constant  $c$  such that  $c > 1/f(x)$  for some  $n_0$ .
- (d) From [Exercise 21](#), we will prove that  $e^{-x} = O(1/g(x))$ . We will also use the result from [Exercise 23, part \(a\)](#) to show the Big-Oh result.

Observe that

$$\begin{aligned} \lim_{x \rightarrow \infty} \frac{e^{-x}}{1/x^3} &= \lim_{x \rightarrow \infty} \frac{x^3}{e^x} \\ &= \lim_{x \rightarrow \infty} \frac{3x^2}{e^x} && \text{(by L'Hôpital's rule)} \\ &= \lim_{x \rightarrow \infty} \frac{6x}{e^x} && \text{(by L'Hôpital's rule)} \\ &= \lim_{x \rightarrow \infty} \frac{6}{e^x} = 0. \end{aligned}$$

Therefore,  $e^{-x} = O(1/x^3) = O(1/g(x))$ . But this implies that  $1/g(x) = \Omega(e^{-x})$ , which completes the proof.  $\square$

**[K] Exercise 23.** Let  $f(n)$  and  $g(n)$  be two positive functions for all  $n \in \mathbb{N}$ .

- (a) Show that, if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ , then  $f(n) = O(g(n))$ .
- (b) Show that, if  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ , then  $f(n) = \Omega(g(n))$ .

*Solution.*

- (a) Suppose that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ . Then, for every  $\epsilon > 0$ , there exist some  $N_\epsilon > 0$  such that, for all  $n > N_\epsilon$ , we have that

$$\left| \frac{f(n)}{g(n)} \right| < \epsilon.$$

Since  $f, g$  are positive functions, this is equivalent to  $f(n) < \epsilon \cdot g(n)$ , which is precisely what it means to say that  $f(n) = O(g(n))$ .

- (b) Suppose that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ . Then, for every  $\epsilon > 0$ , there exist some  $N_\epsilon > 0$  such that, for all  $n > N_\epsilon$ , we have that

$$\left| \frac{f(n)}{g(n)} \right| > \epsilon.$$

Since  $f, g$  are positive functions, this is equivalent to  $f(n) > \epsilon \cdot g(n)$ , which is precisely what it means to say that  $f(n) = \Omega(g(n))$ .

□

**[K] Exercise 24.** Prove the following Big-Oh properties.

- (a) If  $c > 0$  is a constant, then  $c \cdot f(n) = O(f(n))$ . In other words, constants do not affect the growth rate.
- (b) If  $f(n) = O(g(n))$  and  $g(n) = O(h(n))$ , then  $f(n) = O(h(n))$ . In other words, Big-Oh is transitive.
- (c) If  $f_1(n) = O(g_1(n))$  and  $f_2(n) = O(g_2(n))$ , then  $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ .
- (d)  $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$ .

*Solution.*

- (a) This is equivalent to finding constants  $c', n_0 > 0$  such that

$$c \cdot f(n) \leq c' \cdot f(n),$$

for all  $n > n_0$ . Choose  $c' = c$  and  $n_0 = 1$ . Then  $c \cdot f(n)$  is identically  $c' \cdot f(n)$  and the inequality also holds since equality is always true.

- (b) Suppose that  $f(n) = O(g(n))$  and  $g(n) = O(h(n))$ . Since  $f(n) = O(g(n))$ , there exist constants  $c_0, n_0 > 0$  such that

$$f(n) \leq c_0 \cdot g(n),$$

for all  $n > n_0$ . Similarly, since  $g(n) = O(h(n))$ , then there exist constants  $c_1, n_1 > 0$  such that

$$g(n) \leq c_1 \cdot h(n),$$

for all  $n > n_1$ . But this implies that

$$f(n) \leq c_0 \cdot g(n) \leq c_0 (c_1 \cdot h(n)),$$

where this inequality holds whenever  $n > \max\{n_0, n_1\}$ . Choosing  $c = c_0 \cdot c_1$  and  $N = \max\{n_0, n_1\}$ , this shows that  $f(n) = O(h(n))$ .

- (c) Suppose that  $f_1(n) = O(g_1(n))$  and  $f_2(n) = O(g_2(n))$ . Using a similar argument as above, we have constants  $c_1, c_2, n_1, n_2 > 0$  such that

$$f_1(n) \leq c_1 \cdot g_1(n), \quad f_2(n) \leq c_2 \cdot g_2(n),$$

for all  $n > \max\{n_1, n_2\}$ . Under this domain, notice that

$$f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \leq \max\{c_1, c_2\} (g_1(n) + g_2(n)),$$

which gives us the constants and domain for which this inequality holds, proving that  $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ .

- (d) We first show that  $O(f(n) + g(n)) \subseteq O(\max\{f(n), g(n)\})$  and then show the reverse subset inclusion.

Suppose that  $h(n) = O(f(n) + g(n))$ . Then there exist  $c, n_0 > 0$  such that  $h(n) \leq c(f(n) + g(n))$ . But note that this implies that

$$h(n) \leq c \cdot f(n) + c \cdot g(n) \leq c \cdot \max\{f(n), g(n)\},$$

which implies that  $h(n) = O(\max\{f(n), g(n)\})$ . Therefore,  $O(f(n) + g(n)) \subseteq O(\max\{f(n), g(n)\})$

Now suppose that  $h(n) = O(\max\{f(n), g(n)\})$ . Then there exist constants  $c, n_0 > 0$  such that

$$h(n) \leq c \cdot \max\{f(n), g(n)\}.$$

But note that this implies that

$$h(n) \leq c \cdot \max\{f(n), g(n)\} + \min\{f(n), g(n)\} = c(f(n) + g(n)),$$

which implies that  $h(n) = O(f(n) + g(n))$ . This proves both results. □

**[H] Exercise 25.** This exercise shows that Big-Oh is not preserved under monotonic functions. Suppose that  $f(n) = O(g(n))$ , and let  $h(n)$  be some monotonic function in  $n$ .

- (a) Let  $h(n) = 2^n$ . By considering the derivative of  $h(n)$ , or otherwise, show that  $h(n)$  is monotonic in  $n$ .
- (b) Consider  $f(n) = 2^{n+1}$  and  $g(n) = 2^n$ . Show that  $f(n) = O(g(n))$ .
- (c) Show that  $h(f(n)) \neq O(h(g(n)))$ . This shows that Big-Oh is not necessarily preserved under monotonic functions.

*Solution.*

- (a) We can rewrite  $h(n)$  as

$$h(n) = e^{\ln(2^n)} = e^{n \ln 2}.$$

Computing the derivative, we have

$$h'(n) = \ln 2 \cdot e^{n \ln 2} = 2^n \cdot \ln 2.$$

Since  $\ln 2 > \ln 1 > 0$  and the fact that  $2^n > 0$  for all  $n > 0$ , we have that  $h'(n) > 0$  for all  $n$ . Therefore,  $h(n)$  is monotonic in  $n$ .

- (b) We now prove that  $f(n) = O(g(n))$ . But this is clear since we have that

$$f(n) = 2^{n+1} = 2 \cdot 2^n = 2g(n),$$

for all  $n$ . Therefore, choosing  $c = 2$  and  $n = 1$  shows that  $f(n) = O(g(n))$ .

- (c) Consider  $h(f(n))$  and  $h(g(n))$ . We have

$$h(f(n)) = 2^{2^{n+1}}, \quad h(g(n)) = 2^{2^n}.$$

However, it is easy to see that

$$\frac{h(f(n))}{h(g(n))} = \frac{2^{2^{n+1}}}{2^{2^n}} \rightarrow \infty,$$

as  $n \rightarrow \infty$  which means that  $h(f(n)) \neq O(h(g(n)))$ . □

**[E] Exercise 26.** Classify the following pairs of functions by their asymptotic relation; that is, determine if  $f(n) = O(g(n))$ ,  $f(n) = \Omega(g(n))$ , both (i.e.  $f(n) = \Theta(g(n))$ ) or neither.

- (a)  $f(n) = n^{\log n}$ ,  $g(n) = (\log n)^n$ .
- (b)  $f(n) = (-1)^n$ ,  $g(n) = \tan(n)$ .
- (c)  $f(n) = n^{5/2}$ ,  $g(n) = \left[ \log \left( \sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!} \right) \right]^2$ .

*Solution.*

(a) We show that

$$n^{\log n} \ll 2^n \ll (\log n)^n,$$

where  $f(n) \ll g(n)$  is denoted to mean that  $f(n) = O(g(n))$ .

We prove the first half of the inequality; that is,  $n^{\log n} \ll 2^n$ . To see this, we can rewrite both expressions as

$$n^{\log n} = e^{\log(n^{\log n})} = e^{(\log n)^2}, \quad 2^n = e^{\log(2^n)} = e^{n \log 2}.$$

To determine the order of the growth, we compare  $(\log n)^2$  and  $n \log 2$ . By L'Hôpital's Rule, we have that

$$\lim_{n \rightarrow \infty} \frac{(\log n)^2}{n \log 2} = \lim_{n \rightarrow \infty} \frac{2 \log n}{n \log 2}.$$

Since  $\log n < n$  for all  $n > 1$ , the limit is precisely 0. Thus, there exist some  $N > 1$  such that  $(\log n)^2 < n \log 2$  for all  $n > N$ . In other words, we have that

$$e^{(\log n)^2} \ll e^{n \log 2} \implies n^{\log n} \ll 2^n.$$

Now, for sufficiently large  $N$ ,  $\log N > 2$ . So, for all  $n > N$ ,  $2^n \ll (\log n)^n$ . From these two results, it follows that

$$f(n) = O(g(n)).$$

(b) We observe that  $g(n) = \tan(n)$  is  $\pi$ -periodic and passes through  $g(n) = 0$  infinitely many times. Thus, there is no value of  $N$  such that, for *every*  $n > N$ ,  $g(n) > c \cdot f(n)$  or  $g(n) < c \cdot f(n)$ . In other words, it is neither  $O(g(n))$  nor  $\Omega(g(n))$ .

(c) Recall that the Taylor series expansion of  $h(x) = e^x$  is

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Thus, we have that

$$\sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!} = \sum_{k=0}^{\infty} \frac{4^n (4^k n^k)}{k!} = 4^n \sum_{k=0}^{\infty} \frac{(4n)^k}{k!} = 4^n e^{4n}.$$

Then

$$\begin{aligned} g(n) &= \left[ \log \left( \sum_{k=0}^{\infty} \frac{4^{k+n} n^k}{k!} \right) \right]^2 \\ &= (\log(4^n e^{4n}))^2 \\ &= (\log(4^n) + \log(e^{4n}))^2 \\ &= (n \log 4 + 4n)^2 \\ &= c \cdot n^2 \\ &= O(n^{5/2}). \end{aligned}$$

In other words,  $f(n) = \Omega(g(n))$ .

□

**[E] Exercise 27.** Let  $f : \mathbb{N} \rightarrow \mathbb{R}$  be a positive and strictly increasing function; that is, for all  $n \in \mathbb{N}$ ,  $f(n) > 0$ . Show that  $1/f(n) = O(f(n))$ .

*Solution.* Since  $f$  is strictly increasing, then  $f'(n) > 0$  for all  $n \in \mathbb{N}$ . Letting  $g(n) = 1/f(n)$ , we see that

$$g'(n) = -\frac{f'(n)}{[f(n)]^2} < 0.$$

Therefore,  $g(n)$  is a strictly decreasing function. Since  $g$  is decreasing and bounded below by 0, we know that  $g$  converges to a positive number, say  $c$ . However,  $f$  is strictly increasing and unbounded; therefore, for large enough  $n_0$ ,  $f(n) > 1/f(n) \rightarrow c$  for every  $n > n_0$ .  $\square$

**[X] Exercise 28.** Define  $f : \mathbb{N} \rightarrow \mathbb{R}$  by

$$f(n) = \begin{cases} 1 & \text{if } n \leq 2, \\ f\left(\lceil \frac{n}{\log_2 n} \rceil\right) + n & \text{if } n \geq 3. \end{cases}$$

Show that  $f(n) = \Theta(n)$ .

- Show that  $f(n) \leq 2n$  for  $n \geq 512$ .
- Show that  $f(n) \geq n$  for  $n > 2$ .

*Solution.* Throughout the proof, we will omit the ceiling because it is negligible in asymptotic complexity analysis. We show that  $f(n) = \Theta(n)$  by first showing that  $f(n) = O(n)$  and then  $f(n) = \Omega(n)$ . We make the following claim.

**Claim.** For  $n \geq 512$ ,  $f(n) \leq 2n$ .

*Solution.* We proceed with *strong induction*.

The base case is easy to check; we have

$$f(512) = 935 \leq 1024 = 2 \cdot 512.$$

Now assume that  $f(k) \leq 2k$  for all  $512 \leq k < n$ . Now since  $k \geq 512$ , we have that  $\log_2 k \geq \log_2 512 = 9$ . Thus,

$$\frac{n}{\log_2 512} \leq \frac{n}{9}.$$

Then we have that

$$\begin{aligned} f(n) &= f\left(\frac{n}{\log_2 n}\right) + n \\ &\leq f\left(\frac{n}{9}\right) + n && (f \text{ is increasing}) \\ &\leq 2 \cdot \frac{n}{9} + n && (\text{inductive hypothesis}) \\ &= n\left(\frac{2}{9} + 1\right) \\ &\leq 2n. \end{aligned}$$

$\square$

This shows that  $f(n) = O(n)$ . We now show that  $f(n) = \Omega(n)$ . To do this, it is enough to show that  $f(n) \geq n$  for  $n > 2$ . Note that  $n/\log_2 n \geq 1$  for  $n > 2$ . Hence, we have that

$$f(n) = f\left(\frac{n}{\log_2 n}\right) + n \geq f(1) + n = n + 1 > n.$$

Thus,  $f(n) = \Omega(n)$  and combining this result with the previous result, we have that  $f(n) = \Theta(n)$ .  $\square$

# COMP3121/9101

## ALGORITHM DESIGN

### PRACTICE PROBLEM SET 2 – DIVIDE AND CONQUER

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

## Contents

|                                   |    |
|-----------------------------------|----|
| 1 SECTION ONE: RECURRENCES        | 2  |
| 2 SECTION TWO: DIVIDE AND CONQUER | 6  |
| 3 SECTION THREE: QUICKSELECT      | 13 |
| 4 SECTION FOUR: KARATSUBA'S TRICK | 15 |

## § SECTION ONE: RECURRENCES

[K] **Exercise 1.** Determine the asymptotic growth rate of the solutions to the following recurrences. You may use the Master Theorem if it is applicable.

- (a)  $T(n) = 2T(n/2) + O(n)$ .
- (b)  $T(n) = 2T(n/2) + \sqrt{n} + \log n$ .
- (c)  $T(n) = 8T(n/2) + n^{\log_2 n}$ .
- (d)  $T(n) = T(n - 1) + n$ .

*Solution.* (a) Here, we realise that  $a = 2$ ,  $b = 2$ , and  $f(n) = n(2 + \sin n)$ . But  $\sin n \leq 1$  for all  $n$  and so,  $f(n) = \Theta(n)$ .

The critical exponent is

$$c^* = \log_b a = \log_2 2 = 1.$$

Thus, the second case of the Master Theorem applies and we get

$$T(n) = \Theta(n \log n).$$

- (b) Again, we repeat the same process. We realise that  $a = 2$ ,  $b = 2$ , and  $f(n) = \sqrt{n} + \log n$ . So, the critical exponent is  $c^* = 1$ . Since  $\log n$  eventually grows slower than  $\sqrt{n}$ , we have that

$$f(n) = \Theta(\sqrt{n}) = \Theta(n^{1/2}).$$

This implies that

$$f(n) = O(n^{0.9}) = O(n^{c^*-0.1}),$$

so the first case of the Master Theorem applies and we obtain  $T(n) = \Theta(n)$ .

- (c) Here,  $a = 8$ ,  $b = 2$ , and  $f(n) = n^{\log_2 n}$ . So the critical exponent is

$$c^* = \log_b a = \log_2 8 = 3.$$

On the other hand, for large enough  $n$ , we have that  $\log_2 n \geq 4$ . So

$$f(n) = n^{\log_2 n} = \Omega(n^4).$$

Consequently,

$$f(n) = \Omega(n^{c^*+1}).$$

To be able to use the third case of the Master Theorem, we have to show that for some  $0 < c < 1$ , the following holds for sufficiently large  $n$ :

$$af\left(\frac{n}{b}\right) < cf(n).$$

In our case, this translates to

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < cn^{\log_2 n}.$$

Now, we have

$$\begin{aligned}
 8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} &= 8\left(\frac{n}{2}\right)^{\log_2 n - \log_2 2} \\
 &= 8\left(\frac{n}{2}\right)^{\log_2 n - 1} \\
 &< 8\left(\frac{n}{2}\right)^{\log_2 n} \\
 &= \frac{8n^{\log_2 n}}{2^{\log_2 n}} \\
 &= \frac{8}{n}n^{\log_2 n}.
 \end{aligned}$$

If  $n > 16$ , then

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < \frac{1}{2}n^{\log_2 n},$$

so the required inequality is satisfied with  $c = \frac{1}{2}$  for all  $n > 16$ . Therefore, by case 3 of the Master Theorem, the solution is

$$T(n) = \Theta(f(n)) = \Theta(n^{\log_2 n}).$$

- (d) Note that the Master Theorem does not apply; however, we can alter the proof of the Master Theorem to obtain the solution to the recurrence. For every  $k$ , we have  $T(k) = T(k-1) + k$ . So unrolling the recurrence, we obtain

$$\begin{aligned}
 T(n) &= T(n-1) + n \\
 &= \underbrace{[T(n-2) + (n-1)]}_{T(n-1)} + n \\
 &= T(n-2) + (n-1) + n \\
 &= [T(n-3) + (n-2)] + (n-1) + n \\
 &= \dots \\
 &= T(1) + (n - (n-2)) + (n - (n-3)) + \dots + (n-1) + n \\
 &= T(1) + (2+3+\dots+n) \\
 &= T(1) + \frac{n(n+1)}{2} - 1 \\
 &= \Theta(n^2).
 \end{aligned}$$

□

**[K] Exercise 2.** Explain why we cannot apply the Master Theorem to the following recurrences.

- (a)  $T(n) = 2^n T(n/2) + n^n$ .
- (b)  $T(n) = T(n/2) - n^2 \log n$ .
- (c)  $T(n) = \frac{1}{3}T(n/3) + n$ .
- (d)  $T(n) = 3T(3n) + n$ .

*Solution.* (a) The Master Theorem requires the value of  $a$  to be constant. Here, the value of  $a = 2^n$  depends on the input size which is non-constant.

- (b) The Master Theorem requires  $f(n)$  to be non-negative. We can see that, for  $n \in \mathbb{N}$ ,  $f(n) \leq 0$ . So, the Master Theorem cannot be applied.
- (c) The Master Theorem requires  $a \geq 1$ . Here,  $a = 1/3$  and so, the conditions of the Master Theorem are not met.
- (d) The Master Theorem requires  $b > 1$ . However, we see that we can write  $T(n)$  as

$$T(n) = 3T\left(\frac{n}{1/3}\right) + n;$$

in this case, we see that  $b = 1/3 < 1$ . So the conditions of the Master Theorem are not met.  $\square$

**[H] Exercise 3.** Consider the following naive Fibonacci algorithm.

**Algorithm 1**  $F(n)$ : The naive Fibonacci algorithm

---

```

Require:  $n \geq 1$ 
if  $n = 1$  or  $n = 2$  then
    return 1
else
    return  $F(n - 1) + F(n - 2)$ 
end if

```

---

When analysing its time complexity, this yields us with the recurrence  $T(n) = T(n - 1) + T(n - 2)$ . Show that this yields us with a running time of  $\Theta(\varphi^n)$ , where  $\varphi = \frac{1+\sqrt{5}}{2}$  which is the golden ratio. How do you propose that we improve upon this running time?

This is a standard recurrence relation. Guess  $T(n) = a^n$  and solve for  $a$ .

*Solution.* We solve  $T(n) = T(n - 1) + T(n - 2)$ . Using the standard trick of solving recurrence relations, we guess  $T(n) = a^n$  for some  $a$ . Then this produces the recurrence:

$$a^n = a^{n-1} + a^{n-2}.$$

Dividing both sides by  $a^{n-2}$ , we obtain the quadratic equation:

$$a^2 - a - 1 = 0.$$

Then the quadratic equation yields

$$a = \frac{1 \pm \sqrt{5}}{2}.$$

In other words, we obtain

$$T(n) = A \cdot \left(\frac{1 + \sqrt{5}}{2}\right)^n + B \cdot \left(\frac{1 - \sqrt{5}}{2}\right)^n = \Theta(\varphi^n),$$

where  $\varphi = \frac{1+\sqrt{5}}{2}$ .

The inefficiency comes from the fact that we are making unnecessary and repeated calls to problems that we have already seen before. For example, to compute  $F(n)$ , we compute  $F(n - 1)$  and  $F(n - 2)$ . To compute  $F(n - 1)$ , we compute  $F(n - 2)$  again. Because these values never change, a way to improve the overall running time is to *cache* the

results that we have computed previously and only make  $O(1)$  calls for results that we have already computed. This reduces our complexity dramatically from exponential to  $O(n)$ . This is an example of *dynamic programming*, something we will see later down the track.  $\square$

## § SECTION TWO: DIVIDE AND CONQUER

[K] **Exercise 4.** Suppose you have  $n$  sorted lists and suppose that there are  $N$  elements across all  $n$  lists. Note that each list is not necessarily the same length. Design an  $O(N \log n)$  algorithm that merges all  $n$  lists into one sorted list.

For example, if we have two lists  $A_1 = [1, 2]$  and  $A_2 = [1, 4]$ , then the final sorted list should return  $B = [1, 1, 2, 4]$ .

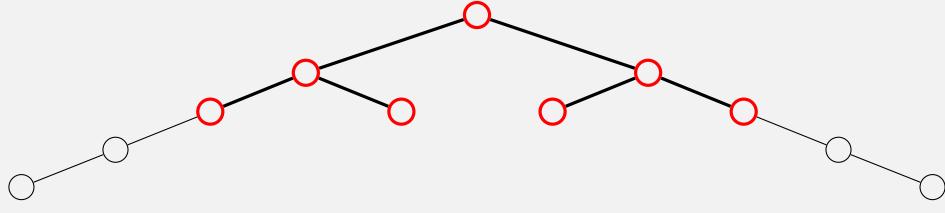
*Solution.* At each recursion, we pair up each of the  $n$  lists to form  $n/2$  pairs. For each pair of lists, we can merge them into a sorted list in  $O(k)$  time, where  $k$  is the number of elements in the final sorted list (i.e. the sum of the number of elements in both lists). At each level, we eventually merge all  $k$  elements in each pair; therefore, in each level, we perform an  $O(N)$  operation merge. Since we are dividing each list into pairs, we have  $\log n$  depth and at each depth, we perform an  $O(N)$  operation. Therefore, the overall algorithm has running time  $O(N \log n)$ .  $\square$

[K] **Exercise 5.** You are given a string  $S$  of length  $n$  and an integer  $k$ . Design an  $O(n^2)$  algorithm that finds the longest substring of  $S$  such that each unique character in the substring has frequency at least  $k$ . For example, if  $S = abbaccc$  and  $k = 2$ , then the longest substring is  $abbaccc$  since the frequency of  $a$  and  $b$  is 2 and the frequency of  $c$  is  $3 \geq k$ .

*Solution.* Note that any character which has a frequency smaller than  $k$  can't be included in our substring. Therefore, we recurse on that particular character and find the longest substring on both sides of the split recursively. This ensures that we never consider a character that we know will never appear in our substring. In the worst case, finding each character takes  $O(n)$  and the maximum depth of our recursion is  $O(n)$  since we could split at every index. Therefore, this gives our algorithm a running time of  $O(n^2)$ .  $\square$

[K] **Exercise 6.** Let  $T$  be a binary tree with  $n$  nodes; we see that a *subtree* of  $T$  is any connected subgraph of  $T$ . A binary tree is *complete* if every internal node has two children, and every leaf node has exactly the same depth. Design an  $O(n)$  algorithm that finds the depth of the *largest complete subtree*.

The following diagram illustrates a binary tree with the largest complete tree highlighted.



*Solution.* We outline each of the three steps of a divide and conquer algorithm for this problem.

- **Divide:** We divide our tree into a left and right subtree. In each subtree, we find the depth of the largest complete subtree as well as the depth of the largest complete subtree rooted at the left and right children respectively. Just finding the depth of the largest complete subtree contained in the left and right subtrees isn't enough for our combine stage to properly work. We need information about the depth at the root of the tree to potentially merge our complete subtrees together.
- **Conquer:** We recursively call on the left and right subtrees, with the base case being the leaves with depth 1 and 1 respectively.
- **Combine:** We now have the largest complete subtree on the left and right subtrees at the root. Let the depth of the largest complete left subtree be  $LCS(L)$  and the depth of the largest complete right subtree be  $LCS(R)$ . Since we have the largest complete subtree at the root of the left and right children respectively, the depth at the

root  $r$  is given by  $LCS(r) = \min(LCS(L), LCS(R)) + 1$ . Our final algorithm then returns the maximum between  $LCS(r)$ , the depth of the largest complete subtree contained entirely in the left subtree, and the depth of the largest complete subtree contained entirely in the right subtree.

To prove that our algorithm is correct, note that, at every recursion, we first find the largest complete tree at the root of the tree. Clearly, the root has depth 1 and we can form a complete subtree by taking the minimum depth among the left and right subtrees that still form a complete subtree. Therefore,  $LCS(r)$  is at least  $1 + \min(LCS(L), LCS(R))$ . To show that  $LCS(r)$  is at most  $1 + \min(LCS(L), LCS(R))$ , we see that removing  $r$  gives a depth of  $LCS(r) - 1$  for the left and right subtree. Among these subtrees, we can find two complete subtrees in the left and right subtrees. But the complete subtree of maximum depth rooted at  $r$  must be the minimum depth among the left and right subtrees. Therefore,  $\min(LCS(L), LCS(R)) \geq LCS(r) - 1$  which implies that  $LCS(r) \leq \min(LCS(L), LCS(R)) + 1$ . The largest complete subtree, therefore, must either be contained entirely on the left, entirely on the right, or at the root. This is exactly what our algorithm returns.

The time complexity is  $O(n)$ , where  $n$  is the number of vertices in our binary tree because we are recursing on every vertex. Therefore, each vertex has  $O(1)$  work, giving our algorithm the running time  $O(n)$ .  $\square$

**[H] Exercise 7.** You and a friend find yourselves on a TV game show! The game works as follows. There is a **hidden**  $n \times n$  table  $A$ . Each cell  $A[i, j]$  of the table contains a single integer and no two cells contain the same value. At any point in time, you may ask the value of a single cell to be revealed. To win the big prize, you need to find the  $n$  cells each containing the **maximum** value in its row. Formally, you need to produce an array  $M[1..n]$  so that  $A[r, M[r]]$  contains the maximum value in row  $r$  of  $A$ , i.e., such that  $A[r, M[r]]$  is the largest integer among  $A[r, 1], A[r, 2], \dots, A[r, N]$ . In addition, to win, you should ask at most  $n \lceil \log N \rceil$  many questions. For example, if the hidden grid looks like this:

|       | Column 1  | Column 2 | Column 3 | Column 4 |
|-------|-----------|----------|----------|----------|
| Row 1 | <b>10</b> | 5        | 8        | 3        |
| Row 2 | 1         | <b>9</b> | 7        | 6        |
| Row 3 | -3        | <b>4</b> | -1       | 0        |
| Row 4 | -10       | -9       | -8       | <b>2</b> |

then the correct output would be  $M = [1, 2, 2, 4]$ .

Your friend has just had a go, and sadly failed to win the prize because they asked  $N^2$  many questions which is too many. However, they whisper to you a hint: they found out that  $M$  is **non-decreasing**. Formally, they tell you that  $M[1] \leq M[2] \leq \dots \leq M[n]$  (this is the case in the example above).

Design an algorithm which asks at most  $O(n \log n)$  many questions that produces the array  $M$  correctly, even in the very worst case.

*Solution.* We first find  $M[N/2]$ . We can do this in  $N$  questions by simply examining each element in row  $N/2$ , and finding the largest one.

Suppose  $M[N/2] = x$ . Then, we know that  $M[i] \leq x$  for all  $i < N/2$  and that  $M[j] \geq x$  for all  $j > N/2$ . Thus, we can recurse to solve the same problem on the sub-grids

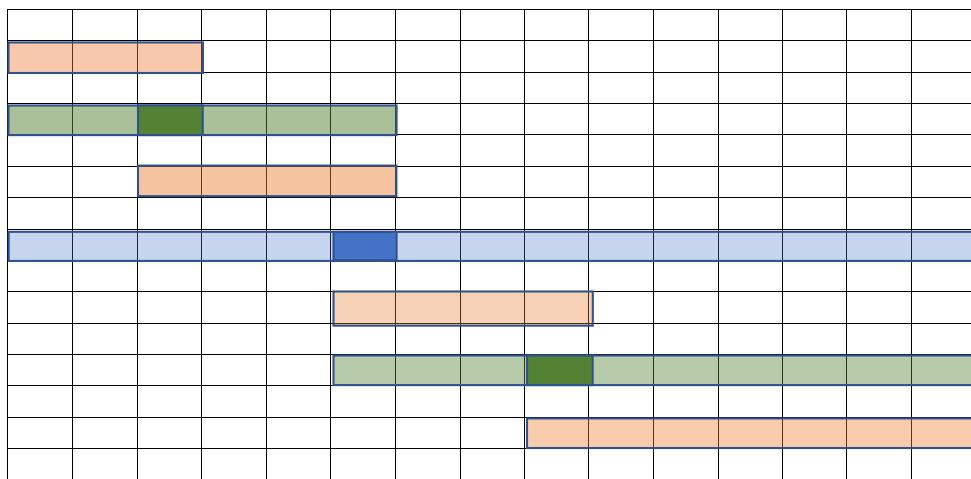
$$A\left[1..\left(\frac{N}{2} - 1\right)\right][1..x]$$

and

$$A\left[\left(\frac{N}{2} + 1\right)..N\right][x..N].$$

It remains to show that this approach uses at most  $2N \lceil \log N \rceil$  questions.

**Claim.** At each stage of recursion, at most two cells of any column are investigated.



*Solution.* We observe the following.

- In the first call of the recursion, only one cell of each column has been investigated. These cells are marked in blue, with the maximum in dark blue.
  - In the second call of the recursion, we investigate two cells of one column, and one cell of each of the other columns. These cells are marked in green, with the maxima in dark green.
  - In the third call of the recursion, we investigate two cells in each of three columns, and one cell of each of the other columns. These cells are marked in orange.

Since the investigated ranges overlap only at endpoints, the claim holds true.

So in each recursion call, at most  $2N$  cells were investigated. The search space decreases by at least half after each call, so there are at most  $\lceil \log_2 N \rceil$  many recursion calls. Therefore, the total number of cells investigated is at most  $2N \lceil \log_2 N \rceil$ .

Additional exercise: using the above reasoning, try to find a sharper bound for the total number of cells investigated.

[H] **Exercise 8.** 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 is as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

- (a) Show, by induction or otherwise, 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) Hence or otherwise, give an algorithm that finds  $F_n$  in  $O(\log n)$  time.

*Solution.* (a) We proceed by induction. If  $n = 1$ , then we have

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

as required.

Let  $n \geq 2$ . Now, suppose that

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

Then

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} F_n + F_{n-1} & F_{n-1} + F_{n-2} \\ F_{n-1} + F_{n-2} & F_{n-2} + F_{n-3} \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad (\text{inductive hypothesis}) \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n. \end{aligned}$$

This finishes the induction proof.

(b) Let  $G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ . We can compute  $G^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2$  in  $O(1)$  since each element in the matrix is just a sum and product of small numbers. Likewise, we can also compute powers of  $G$  that are powers of two; in other words, we compute the matrices:  $G, G^2, G^4, \dots$

To determine  $F_n$ , we need to find which combination of  $G^{2^\ell}$  matrices we want to take. Thus, we can write  $n$  in its binary representation; that is, write

$$n = a_0 2^0 + a_1 2^1 + a_2 2^2 + \dots$$

Each  $a_i$  term tells us precisely which matrices of the form  $G^{2^i}$  we want to multiply to obtain a final matrix  $H$ . The top right (or bottom left) of  $H$  then tells us precisely what  $F_n$ . To argue the time complexity, we observe that there are maximally  $\lfloor \log_2 n \rfloor$  to consider when writing down the binary representation of  $n$ . Therefore, we are performing  $\log n$  many constant time operations, which give us  $O(\log n)$  time complexity. □

**[E] Exercise 9.** Let  $A$  be an array of  $n$  integers, not necessarily distinct or positive. Design a  $\Theta(n \log n)$  algorithm that returns the maximum sum found in any contiguous subarray of  $A$ .

*Solution.* Note that brute force takes  $\Theta(n^2)$ .

The key insight to this problem is the following: once we split the original array in half, the maximum subarray must occur in one of the following cases:

- the maximum subarray sum occurs *entirely* in the left half of the original array.
- the maximum subarray sum occurs *entirely* in the right half of the original array.
- the maximum subarray sum overlaps across both halves.

We can compute the first two cases simply by making recursive calls on the left and right halves of the array. The third case can be computed in linear time. We now fill in the details of the algorithm.

Split the array into the left half and right half at the midpoint  $m$ .

- We can compute the maximum subarray sum in the left half using recursive calls on the left half of the array  $A[1..m]$ .
- Similarly, we can compute the maximum subarray sum in the right half using recursive calls on the right half of the array  $A[(m+1)..n]$ .
- To compute the maximum sum in the case where the subarray overlaps across both halves, we think of such a subarray as containing a portion from the left half and a portion from the right half. Now, we
  - find the maximum sum of any subarray of the form  $A[l..m]$  by scanning left from  $m$ , and
  - similarly maximise the sum  $A[(m+1)..r]$  by scanning right from  $m+1$ .

Adding these two results gives the maximum sum of a subarray of the form  $A[l..r]$  where  $l \leq m < r$  as required.

- Our algorithm then returns the maximum of these three cases.
  - One caveat is that, if the maximum sum is negative, then our algorithm should return 0 to indicate that choosing an *empty* subarray is best.

To compute the time complexity of the algorithm, note that we're splitting our problem from size  $N$  to two  $N/2$  problems and doing an  $O(n)$  overhead to combine the solutions together. This gives us the recurrence relation

$$T(n) = 2T(n/2) + O(n).$$

This falls under Case 2 of the Master Theorem, so  $T(n) = \Theta(n \log n)$  is the total running time of the algorithm.  $\square$

**[E] Exercise 10.** Suppose 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.

Compare the numbers in pairs; how many potential candidates can exist?

*Solution.* Create two arrays  $B$  and  $C$ , initially both empty. Repeat the following procedure  $n$  times:

- Pick any two elements of  $A$  and compare them.
  - If the numbers are different, discard both of them.
  - If instead the two numbers are equal, append one of them to  $B$  and the other to  $C$ .

**Claim.** If a value  $x$  appears in at least half the entries of  $A$ , then the same value also appears in at least half the entries of  $B$ .

*Proof.* Suppose such a value  $x$  exists. In a pair of distinct numbers, at most one of them can be  $x$ , so after discarding both,  $x$  still makes up at least half the remaining array elements. Repeating this,  $x$  must account for at least half the values appearing in  $B$  and  $C$  together. But  $B$  and  $C$  are identical, so at least the entries of  $B$  are equal to  $x$ .

We can now apply the same algorithm to  $B$ , and continue reducing the instance size until we reach an array of size two. The elements of this array are the only candidates for the original property, i.e., the only values that could have appeared  $n$  times in the original array.

There is a special case;  $B$  and  $C$  could be empty after the first pass. In this case, the only candidates for the property are the values which appear in the last pair to be considered. For each of these, perform a linear scan to find their frequency in the original array, and report the answer accordingly.

Finally we analyse the time complexity. Clearly, each step of the procedure repeated  $n$  times above takes constant time, so we can reduce the problem from array  $A$  (of length  $2n$ ) to array  $B$  (of length  $\leq n$ ) in  $\Theta(n)$  time. Letting the instance size be *half* the length of the array (to more neatly fit the Master Theorem), we can express the worst case using the recurrence

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

Now  $a f(n/b) = f(n/2)$ , which is certainly less than say  $0.9f(n)$  for large  $n$  as  $f$  is linear. By case 3 of the Master Theorem, the time complexity is  $T(n) = \Theta(f(n)) = \Theta(n)$ , as required.  $\square$

**[E] Exercise 11.** You are given  $n$  points on a plane. The  $i$ th point is labelled with coordinates  $(x_i, y_i)$ . Design an  $O(n \log n)$  algorithm that finds the pair of points that is closest together, where “closeness” is measured by its Euclidean distance.

*Solution.* Given our set of points, we sort our points by its  $x$ -coordinate and find the middle point – say  $a$  – to divide our set into two parts. During the pre-processing stage, we also sort our points by its  $y$ -coordinate to ensure that we have a linear-time comparison later on.

This gives all the points whose  $x$ -coordinate is smaller than the  $x$ -coordinate of  $a$  in one list, and all points whose  $x$ -coordinate is larger than the  $x$ -coordinate of  $a$  in the other list. From these two smaller lists, we have one of three situations:

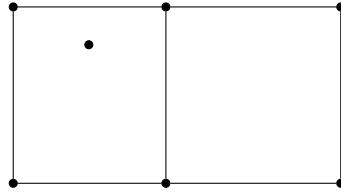
- The closest pair of points are situated *entirely* inside the left list;
- The closest pair of points are situated *entirely* inside the right list;
- The closest pair of points have one point in the left list and one point in the right list.

The first two subproblems can be solved recursively on the list of points that are situated entirely on the left and right sides respectively. We now manage the final situation. From the first two subproblems, let  $\delta_L$  be the distance of the closest pair of points on the left subproblem and  $\delta_R$  be the distance of the closest pair of points on the right subproblem. Define  $\delta = \min\{\delta_L, \delta_R\}$ . Clearly, the closest pair of points where one point is on the left and one point is on the right must have distance at most  $\delta$ ; otherwise, the pair of points whose distance is  $\delta$  are closer.

Therefore, any pair of points whose distance is at most  $\delta$  must be contained within a  $2\delta$ -width rectangle centred around the dividing line (i.e. the  $x$ -coordinate of  $a$ ). This reduces our search space to looking at points within a rectangle of width  $2\delta$ . We now need to narrow down our search space even more since the height of our current rectangle is unbounded.

Consider any point  $b$  inside our current strip. To check if there is some point whose distance is smaller than  $\delta$ , we need to ensure that it is also within a  $\delta$ -distance vertically. Therefore, we consider any point inside a  $\delta \times 2\delta$  rectangular slab containing  $b$ . We now claim that there are *at most* 6 points inside this rectangle.

To see why this is true, consider the following placement of points.



These are the *extremities* of point placement because any seventh point implies that there are two points in exactly one side whose distance is necessarily smaller than  $\delta$ . But this is a contradiction because we would have found this earlier in our algorithm. Therefore, for *each* point in our  $\delta \times 2\delta$  rectangle, we perform at most 6 comparisons to update our minimum by finding all points that is vertically within  $\delta$  distance in the other half of our rectangle. In other words, the merging process of our algorithm has running time  $O(n)$ .

This gives our recurrence

$$T(n) = 2T(n/2) + O(n),$$

which gives  $T(n) = \Theta(n \log n)$  by the Master Theorem. □

## § SECTION THREE: QUICKSELECT

[K] **Exercise 12.** Consider  $n$  points on a plane. The  $i$ th point has coordinates  $(x_i, y_i)$ . You are also given an integer  $k$ .

- Design an  $O(n \log n)$  algorithm that finds the  $k$  closest points from the origin, where “closeness” is measured by its Euclidean distance from the origin. In other words, we say that point  $i$  is  $\sqrt{x_i^2 + y_i^2}$  distance away from the origin. You may choose to return the  $k$  closest points in any order.
- Design an algorithm with expected running time  $O(n)$  that solves the same problem.

There is an easy way to solve part (a); how might you optimise your strategy given that you can return your solution in *any* order for part (b)?

*Solution.*

- For each point  $i$ , compute its distance from the origin  $x_i^2 + y_i^2$  (why is it enough to look simply at  $x_i^2 + y_i^2$ ?) in constant time and store the distance in an array  $A$ . We can then sort the array by MERGESORT with  $O(n \log n)$  running time. The  $k$  closest points are then the first  $k$  points in  $A$  (i.e.  $A[1], A[2], \dots, A[k]$  after sorting).
- This is a classical problem that can be solved with QUICKSELECT. Firstly, we can compute each distance in constant time and store the distances in an array  $A$ . Therefore, we may assume that the array  $A$  contains all of the distances for each corresponding point. From this, we can now use QUICKSELECT to find the  $k$  smallest elements in  $A$ . This has expected running time  $O(n)$  due to QUICKSELECT and every other operation is linear in the worst case. Therefore, the expected running time is  $O(n)$ .

□

[K] **Exercise 13.** Suppose that we modify the median of medians QUICKSELECT algorithm to use blocks of 7 rather than 5. Determine a worst-case recurrence, and solve for the asymptotic growth rate.

*Solution.* We first recurse to find the median of the  $n/7$  block medians. That median is guaranteed to be greater than four elements in each of  $n/14$  blocks, and less than four elements in each of  $n/14$  blocks. Therefore the worst case partition is in a ratio of 2 : 5, and we may have to recursively select from  $5n/7$  of the original items. Therefore the recurrence is

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + \Theta(n).$$

Now we can deduce a pattern that arises from the number of items:

- the 2 subproblems at the second level of the recursion have a total of  $6n/7$  items,
- the 4 subproblems at the third level of the recursion have a total of  $36n/49$  items,
- and so on.

The total size of all subproblems is bounded above by

$$n \left( 1 + \frac{6}{7} + \left( \frac{6}{7} \right)^2 + \dots \right) = \frac{n}{1 - \frac{6}{7}} = 7n,$$

so the total time taken (for forming and sorting blocks as well as partitioning) is  $\Theta(n)$ .

□

**[K] Exercise 14.** Given two *sorted* arrays  $A$  and  $B$ , each of length  $n$ , design a  $O(\log n)$  algorithm to find the (lower) median of all  $2n$  elements of both arrays. You may assume that  $n$  is a power of two and that all  $2n$  values are distinct.

*Solution.* If  $A[n/2] < B[n/2]$ , then  $A[n/2]$  is smaller than the median, and  $B[n/2 + 1]$  is larger than the median. We can therefore discard the first half of  $A$  and the second half of  $B$ , and the median of the remaining  $n$  elements is our answer.

Similarly, if  $A[n/2] > B[n/2]$ , we instead discard the second half of  $A$  and the first half of  $B$ . We have transformed an instance of size  $n$  to an instance of size  $n/2$  using only one comparison, so the recurrence is

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

The critical exponent is  $c^* = \log_2 1 = 0$ , so by Case 2 of the Master Theorem, the asymptotic solution is  $T(n) = \Theta(\log n)$ . Note that this is the same recurrence as binary search.  $\square$

**[K] Exercise 15.** Given an array  $A$  of  $n$  distinct integers and an integer  $k < n/2$ , design an  $O(n)$  algorithm which finds the median as well as the  $k$  values closest to the median from above and below. You may assume that  $n$  is odd.

*Solution.* Apply median of medians quickselect to find the values  $k$  ranks below and  $k$  ranks above the median (i.e. the  $(\frac{n+1}{2} - k)$ th and  $(\frac{n+1}{2} + k)$ th smallest values. Then scan through the array, checking whether each element lies between these bounds.  $\square$

**[E] Exercise 16.** Suppose that we are given a set  $S$  of  $n$  elements. Each element has a value and a weight. For an element  $x \in S$ , let

- $S_{<x}$  be the set of elements whose value is less than the value of  $x$ ,
- $S_{>x}$  be the set of elements whose value is greater than the value of  $x$ .

Let  $R \subseteq S$  be a subset of  $S$  and define its weight  $w(R)$  to be the sum of the weights of the elements in  $R$ . The *weighted median* of  $S$  is any element  $x$  such that  $w(S_{<x}) \leq w(S)/2$  and  $w(S_{>x}) \leq w(S)/2$ .

Design an  $O(n)$  algorithm that computes the weighted median of  $S$ .

*Solution.* This effectively uses the same median of median quickselect algorithm with a slight modification. Once we partition on  $x$ , we compute the sum of the weights of all elements in  $S_{<x}$  (i.e.  $w(S_{<x})$ ) and compute the sum of the weights of all elements in  $S_{>x}$  (i.e.  $w(S_{>x})$ ). If both are greater than  $1/2$ , then  $x$  is the weighted median of  $S$ . Otherwise, one of them is greater than  $1/2$  and we recurse on that side. Note that the additional computation takes linear time and so, it doesn't worsen the overall time complexity of our algorithm.  $\square$

## § SECTION FOUR: KARATSUBA'S TRICK

[K] **Exercise 17.** Let  $P(x) = a_0 + a_1x$  and  $Q(x) = b_0 + b_1x$ , and define  $R(x) = P(x) \cdot Q(x)$ . Find the coefficients of  $R(x)$  using only three products of pairs of expressions each involving the coefficients  $a_i$  and/or  $b_j$ .

Addition and subtraction of the coefficients do not count towards this total of three products, nor do scalar multiples (i.e. products involving a coefficient and a constant).

*Solution.* We simply use Karatsuba's trick.

Note that

$$\begin{aligned} R(x) &= (a_0 + a_1x)(b_0 + b_1x) \\ &= a_0b_0 + (a_0b_1 + a_1b_0)x + a_1b_1x^2 \\ &= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1] + a_1b_1x^2. \end{aligned}$$

Therefore, we have three multiplications:  $a_0b_0$ ,  $a_1b_1$ , and  $(a_0 + a_1)(b_0 + b_1)$ .  $\square$

[K] **Exercise 18.** Recall that the product of two complex numbers is given by

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc).$$

- (a) Multiply two complex numbers  $a + bi$  and  $c + di$ , where  $a, b, c, d$  are real numbers, using only three multiplications.
- (b) Find  $(a + ib)^2$  using only two multiplications.
- (c) Find the product  $(a + bi)^2(c + di)^2$  using only five multiplications.

*Solution.*

- (a) We see that

$$\begin{aligned} (a + bi)(c + di) &= (ac - bd) + i(ad + bc) \\ &= (ac - bd) + i[(a + b)(c + d) - ac - bd]. \end{aligned}$$

Therefore, we only need three multiplications:  $ac$ ,  $bd$ ,  $(a + b)(c + d)$ .

- (b) Expanding and simplifying, we have that

$$\begin{aligned} (a + ib)^2 &= a^2 - b^2 + 2abi \\ &= (a - b)(a + b) + (a + a)bi. \end{aligned}$$

We only require two multiplications here:  $(a - b)(a + b)$ ,  $(a + a)b$ .

- (c) We first see that

$$(a + bi)^2(c + di)^2 = [(a + bi)(c + di)]^2.$$

Using the previous part, computing the square can be done using two multiplications. From part (a), we could compute  $(a + bi)(c + di)$  using three multiplications. Therefore, we first compute the three multiplications to find  $(a + bi)(c + di)$  and then the remaining two multiplications to compute its square. This gives five multiplications.  $\square$

**[K] Exercise 19.** Let  $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$  and  $Q(x) = b_0 + b_1x + b_2x^2 + b_3x^3$ , and define  $R(x) = P(x) \cdot Q(x)$ . You are tasked to find the coefficients of  $R(x)$  using only twelve products of pairs of expressions each involving the coefficients  $a_i$  and/or  $b_j$ .

Can you do better?

*Solution.* We use Karatsuba's trick. We see that

$$\begin{aligned} R(x) &= (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3) \\ &= (a_0 + a_1x)(b_0 + b_1x) + (a_0 + a_1x)(b_2 + b_3x)x^2 \\ &\quad + (a_2 + a_3x)(b_0 + b_1x)x^2 + (a_2 + a_3x)(b_2 + b_3x)x^4 \\ &= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]x + a_1b_1x^2 \\ &\quad + (a_0b_2 + [(a_0 + a_1)(b_2 + b_3) - a_0b_2 - a_1b_3]x + a_1b_3x^2)x^2 \\ &\quad + (a_2b_0 + [(a_2 + a_3)(b_0 + b_1) - a_2b_0 - a_3b_1]x + a_3b_1x^2)x^2 \\ &\quad + (a_2b_2 + [(a_2 + a_3)(b_2 + b_3) - a_2b_2 - a_3b_3]x + a_3b_3x^2)x^4. \end{aligned}$$

Note that each of the last four lines involves only three multiplications. Expanding and regrouping will yield the coefficients of  $R(x)$ .  $\square$

# COMP3121/9101

## ALGORITHM DESIGN

### PRACTICE PROBLEM SET 3 – GREEDY ALGORITHMS

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

## Contents

|                                  |    |
|----------------------------------|----|
| 1 SECTION ONE: ORDERING          | 2  |
| 2 SECTION TWO: SELECTION         | 9  |
| 3 SECTION THREE: SCHEDULING      | 16 |
| 4 SECTION FOUR: GAMES AND GRAPHS | 22 |

## § SECTION ONE: ORDERING

### [K] Exercise 1.

- (a) There are  $n$  robbers who have stolen  $n$  items. You would like to distribute the items among the robbers (one item per robber). You know the precise value of each item. Each robber has a particular range of values they would like their item to be worth (too cheap and they will not have made money, too expensive and they will draw a lot of attention). Devise an algorithm that runs in  $O(n^2)$  which either produces the best distribution such that every robber is happy or determines that there is no such distribution.
- (b) Using an exchange argument, justify the correctness of your algorithm.
- Start by supposing that there is some other solution that assigns the items to the robbers that makes all robbers happy, which is different to your greedy solution.
  - By choosing suitable items, how do you justify that swapping the items between your solution and the chosen solution will not worsen your solution?
  - Argue that, by repeatedly swapping your items, you can transform any arbitrarily correct solution into yours.

*Solution.*

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

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

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

Now suppose this robber was assigned an item  $m$  with value  $v_m$  (so  $v_m \in [L_j, U_j]$ ). Since item  $k$  is the least value item for which our greedy strategy was violated,  $v_k < v_m$ . Hence,  $v_m \in [L_i, U_i]$ , which means that robber  $i$  would be happy with item  $m$ , and we can therefore swap the assignments for the two robbers while keeping them both happy. By repeatedly swapping assignments that violate our greedy strategy in this manner, we eventually reach an assignment that adheres to our strategy. Therefore, our method is optimal.

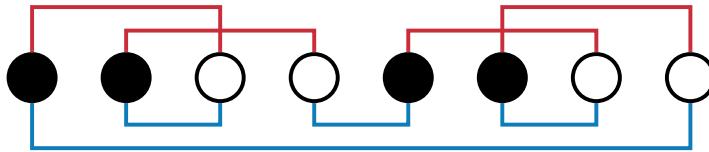
□

### [K] Exercise 2.

- (a) Assume that you are given  $n$  white and  $n$  black dots lying in a random configuration on a straight line, equally spaced. Design a greedy algorithm which connects each black dot with a (different) white dot, so that the total length of wires used to form such connected pairs is minimal. The length of wire used to connect two dots is equal to the straight line distance between them.
- (b) Justify the correctness of your algorithm by using an exchange argument.

*Solution.*

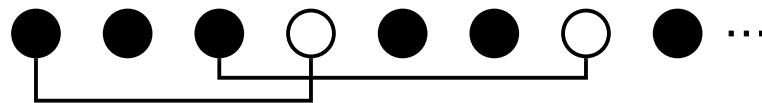
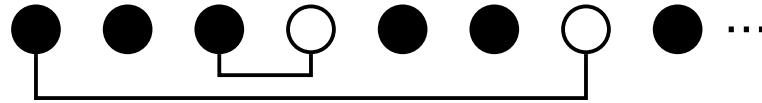
- (a) One should be careful about what kind of greedy strategy one uses. For example, connecting the closest pairs of equally coloured dots produces suboptimal solution as the following example shows:



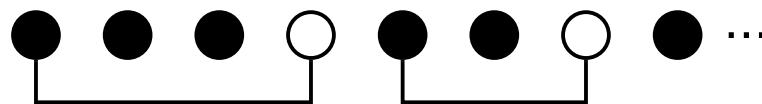
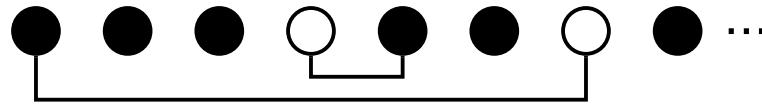
Connecting the closest pairs (blue lines) uses  $3 + 7 = 10$  units of length while the connections in red use only  $4 \times 2 = 8$  units of length.

The correct approach is to go from left to right and connect the leftmost dot with the leftmost dot of the opposite colour and then continue in this way. By storing pointers to the leftmost dots and updating it as we go, the overall complexity of our algorithm is  $O(n)$ .

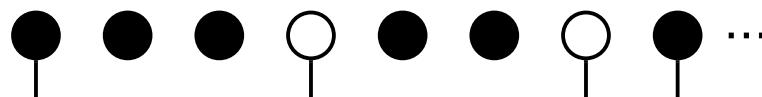
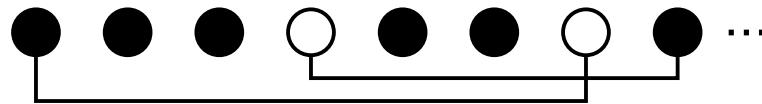
- (b) To prove that this is optimal, we assume that there is a different strategy which uses less wire for a particular configuration of dots. Such a strategy will disagree with our greedy strategy at least once, which means that at some point, it will not connect the leftmost available dot with the leftmost available dot of the opposite colour. We look at the leftmost dot for which the greedy strategy is violated. There are three types of configurations to consider (shown in the figure below), but for each configuration, the strategy uses either the same amount or more wire than the greedy strategy. Hence, the hypothetical strategy is no better than the greedy strategy, contradicting our original assumption, and so the greedy strategy is optimal.



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



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



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

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

□

**[K] Exercise 3.** Assume that you got a fabulous job and you wish to repay your student loan as quickly as possible. Unfortunately, the bank Western Road Robbery which gave you the loan has the condition that you must start your monthly repayments by paying off \$1 and then each subsequent month you must pay either double the amount you paid the previous month, the same amount as the previous month or half the amount you paid the previous month. On top of these conditions, your last payment must be \$1.

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

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

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

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

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

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

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

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

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

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

Therefore, we restrict ourselves to schedules of the form

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

where bracketed terms are optional.

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

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

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

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

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

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

Now we can present the algorithm.

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

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

**[H] Exercise 4.** After the success of your latest research project in mythical DNA, you have gained the attention of a most diabolical creature: Medusa. Medusa has snakes instead of hair. Each of her snakes' DNA is represented by an uppercase string of letters. Each letter is one of S, N, A, K or E. Your extensive research shows that a snake's venom level depends on its DNA. A snake has venom level  $x$  if its DNA:

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

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

Combine greedy with binary search.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## § SECTION TWO: SELECTION

[K] **Exercise 5.** Consider the following problem.

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

Consider the following greedy algorithm.

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

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

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

[K] **Exercise 6.**

- (a) There are  $n$  people that you need to guide through a difficult mountain pass. Each person has a speed in days that it will take to guide them through the pass. You will guide people in groups of up to  $k$  people, but you can only move as fast as the slowest person in each group. Design an  $O(n \log n)$  algorithm to determine the fewest number of days you will need to guide everyone through the pass.
- (b) Justify the correctness of your argument using an exchange argument.

*Solution.*

- (a) Create a group consisting of the  $K$  slowest people, another group consisting of the next  $K$  slowest people, and so on. Note that the final group consisting of the fastest people may contain fewer than  $K$  people. Then guide these groups one by one through the pass (the order in which you guide the groups does not matter). As we only need to sort the array of people and then go through them once, the total complexity is  $O(n \log n)$ .
- (b) The optimality of this strategy should be obvious. Consider the slowest person. This person will need to be guided through the mountain pass eventually, and since you can only move as fast as the slowest person in each group, the speed of the other people in the group will not affect the speed of the group. So to save as much time as possible, you should send them with as many of the other slowest people as possible. The same argument applies to the remaining people after this group has been sent. Formally, suppose that some optimal solution  $O$  that disagrees with our greedy solution  $G$  (here we assume that Groups of  $O$  is sorted by the speed of the slowest person), we may look at the first instance where a person belongs to a different group in  $O$  ( $p_i$ ) than  $G$  ( $p_j$ ). Note that it cannot be the case that this person is the slowest person in the group else it violates the assumption that  $O$  is optimal. Then, we can choose to again swap the spots of  $p_i$  with  $p_j$  in  $O$  without losing optimality ( $p_i > p_j$  in terms of speed). Therefore, our exchange argument is complete.

$\square$

[K] **Exercise 7.**

- (a) There are  $n$  courses that you can take. Each course has a minimum required IQ, and will raise your IQ by a certain amount when you take it. Design an  $O(n^2)$  algorithm to find the minimum number of courses you will need to take to get an IQ of at least  $K$ .
- (b) Justify the correctness of your algorithm using a greedy stays ahead argument.

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

*Solution.*

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

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

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

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

□

### [K] Exercise 8.

- (a) You have  $n$  items for sale, numbered from 1 to  $n$ . Alice is willing to pay  $a[i] > 0$  dollars for item  $i$ , and Bob is willing to pay  $b[i] > 0$  dollars for item  $i$ . Alice is willing to buy no more than  $A$  of your items, and Bob is willing to buy no more than  $B$  of your items. Additionally, you must sell each item to either Alice or Bob, but not both, so you may assume that  $n \leq A + B$ . Given  $n, A, B, a[1..n]$  and  $b[1..n]$ , you have to determine the **maximum** total amount of money you can earn in  $O(n \log n)$  time.
- (b) Justify the correctness of your algorithm using a greedy stays ahead argument.

*Solution.*

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

Computing all the differences  $d[i] = |a[i] - b[i]|$  takes  $O(n)$  time, sorting  $d[i]$ 's takes  $O(n \log n)$  time and going through the list lastly takes  $O(n)$  time. Thus the total run time complexity is  $O(n \log n)$ .

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

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

□

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

*Solution.* To solve this problem, we proceed to keep giving the coin with the largest denomination that is less than or equal to the amount remaining until the desired amount is reached.

To prove that this results in the smallest possible number of coins for any amount to be paid, we assume the opposite - that is, suppose that for a certain amount  $M$ , there is an optimal way of payment that is more efficient than the one described by the greedy algorithm. Since the ordering in which the coins are given does not matter, we can assume that such payment proceeds from the largest to the smallest denomination.

Consider the instance of the greedy policy being violated. This can happen, for example, if the remaining amount to be paid is at least \$2, but a \$2 coin was not used. However, if this is the case, notice that at most one \$1 coin could have been used, as otherwise, the payment would not be efficient because two \$1 coins can be replaced by a single \$2 coin. Thus, after the option of giving a \$1 coin has been exhausted we are left with at least \$1 to be given without using any \$1 coins. For the same reason at most one 50c coin can be given and we are left with at least 50 cents to be given without using any 50c coins. Note that at most two 20c coins can be used because three 20c coins can be replaced with a 50c and 10c coin. Also note that if two 20c coins are used, no 10c coins can be used because two 20c coins and a 10c coin can be replaced with a single 50c coin.

Thus, if two 20c coins are used, only 5c coins can be used to give the remaining amount of at least 10 cents, which would require two 5c coins, but these could be replaced by a single 10c coin, contradicting optimality. If, on the other hand, only one 20c coin is used to give an amount of at least 50 cents we are left with at least 30 cents to be given using 10c and 5c coins. Note that in such a case only one 10c coin can be used because two 10c coins can be replaced by a 20c coin. So we are left an amount of at least 20 cents to be given with 5c coins only. However, only one 5c coin can be used because two 5c coins can be replaced by one 10c coin contradicting the optimality of the solution. If the greedy strategy was violated for the first time when smaller amounts are due, the analysis is a subset of the analysis above. Thus, the greedy strategy provides an optimal solution. Note that in all countries the notes and coins denominations are chosen so that the greedy strategy is optimal.

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

**[K] Exercise 10.** Assume that the denominations of your  $n + 1$  coins are  $1, c, c^2, c^3, \dots, c^n$  for some integer  $c > 1$ . Design a greedy algorithm that runs in linear time complexity for which, given any amount, makes that amount using a minimal number of coins.

How does this problem differ to Exercise 21?

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

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

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

**[K] Exercise 11.** Given two sequences of letters  $A$  and  $B$ , find if  $B$  is a sub-sequence of  $A$  in the sense that one can delete some letters from  $A$  and obtain the sequence  $B$ . Then prove that your algorithm is correct by either using the greedy stays ahead or exchange argument.

*Solution.* Use a simple greedy strategy. First, find and mark the earliest occurrence of the first letter of  $B$  in  $A$ . Then, for each subsequent letter of  $B$ , find and mark the earliest occurrence of that letter in  $A$  which is after the last marked letter. If you reach the end of  $B$  before or at the same time as you reach the end of  $A$ , then  $B$  is a sub-sequence of  $A$ .

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

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

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

How is this similar to INTERVAL STABBING?

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

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

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

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

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

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

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

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

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

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

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

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

**[E] Exercise 14.** Let  $A$  be a  $2 \times n$  array of positive real numbers such that the elements in each column sum to 1. Design an  $O(n \log n)$  algorithm to pick one number in each column such that the sum of the numbers chosen in each row never exceeds  $\frac{n+1}{4}$ .

For example, consider the following  $2 \times 8$  array.

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

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

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

The chosen numbers on the first row give us

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

while the chosen numbers on the second row give us

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

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

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

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

but

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

which completes the proof. □

## § SECTION THREE: SCHEDULING

[K] **Exercise 15.** Let  $X$  be a set of  $n$  intervals on the real line. A subset of intervals  $Y \subseteq X$  is called a tiling path if the intervals in  $Y$  cover the intervals in  $X$ , that is, any real value that is contained in some interval in  $X$  is also contained in some interval in  $Y$ . The size of a tiling cover is just the number of intervals. Describe and analyse an algorithm to compute the smallest tiling path of  $X$  in  $O(n^2)$  time. Assume that your input consists of two arrays  $L[1..n]$  and  $R[1..n]$ , representing the left and right endpoints of the intervals in  $X$ .



Problem Set 3/Section 3/tiling\_path.pdf

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

*Solution.* Sort the intervals in increasing order of their left endpoints. Start with the interval with the smallest left endpoint; if there are several intervals with the same such smallest left endpoint choose the one with the largest right endpoint. Then, consider all intervals whose left endpoints are in the last chosen interval, and pick the one with the largest right endpoint (as long as this right endpoint is beyond the right endpoint of the last chosen interval). If there are no such intervals, move on to the next smallest left endpoint and pick again as in the first step. Continue in this manner until an interval with the absolute largest right endpoint is chosen. Note that the right endpoints of the chosen intervals also form an increasing subsequence of  $R$ .

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

- $L(g_i)$  must be greater than or equal to  $L(g_{i-1})$  by the sort order, and furthermore they must be unequal as otherwise the greedy algorithm would only have taken one of these two intervals. Therefore  $L(g_i) > L(g_{i-1})$ .
- If  $L(g_i) \leq R(g_{i-1})$ , i.e. interval  $g_i$  has left endpoint within interval  $g_{i-1}$ , then  $R(g_i) > R(g_{i-1})$ , as otherwise the greedy algorithm would not have taken interval  $g_i$ .

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

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

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

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

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

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

**[K] Exercise 16.** A photocopying service with a single large photocopying machine faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to schedule the jobs on their single machine in an order that keeps their customers the happiest. Customer  $i$ 's job will take  $t_i$  time to complete. Given a schedule (i.e., an ordering of the jobs), let  $C_i$  denote the finishing time of job  $i$ . For example, if job  $i$  is the first to be done we would have  $C_i = t_i$ , and if job  $j$  is done right after job  $i$ , we would have  $C_j = C_i + t_j$ . Each customer  $i$  also has a given weight  $w_i$  which represents his or her importance to the business. The happiness of customer  $i$  is expected to be dependent on the finishing time of their job. So the company decides that they want to order the jobs to minimise the weighted sum of the completion times,  $\sum_{i=0}^n w_i C_i$ . Design an  $O(n \log n)$  algorithm to solve this problem. That is, you are given a set of  $n$  jobs with a processing time  $t_i$  and a weight  $w_i$  for job  $i$ . You want to order the jobs so as to minimise the weighted sum of the completion times,  $\sum_{i=0}^n w_i C_i$ .

*Solution.* Schedule the jobs in decreasing order of  $w_i/t_i$ . This problem is very similar to the Tape Storage problem, which was covered in the lectures. To prove that this is optimal, we first introduce the concept of an inversion. We say that two jobs  $i$  and  $j$  form an inversion if job  $i$  is scheduled before job  $j$ , but  $w_i/t_i < w_j/t_j$ . Now consider any schedule which violates our greedy scheduling. Such a schedule will contain an inversion of at least one pair of consecutive jobs. If we repeatedly fix all such inversions between two consecutive jobs (by swapping them in the schedule) without increasing the value of  $S = \sum_{i=0}^n w_i C_i$  then, by the “bubble sort algorithm” argument, all inversions will eventually disappear and we will have shown that the greedy solution is no worse than the solution considered. So let us prove that swapping two consecutive inverted jobs can only reduce the value of  $\sum_{i=0}^n w_i C_i$ . Assume that we have re-enumerated the jobs so that they are numbered according to their place in the schedule, so that the two inverted jobs are numbered

$i$  and  $i + 1$ . Now let  $T$  be the time just before job  $i$  starts. Note that by swapping two successive jobs only two terms of the sum  $\sum_{i=0}^n w_i C_i$  change: before the swap this sum contains  $w_i(T + t_i) + w_{i+1}(T + t_i + t_{i+1})$ , while after the swap the new sum  $S'$  will contain  $w_{i+1}(T + t_{i+1}) + w_i(T + t_{i+1} + t_i)$ . Thus,

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

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

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

**[K] Exercise 17.** You have  $n$  students with varying skill levels and  $n$  jobs with varying skill requirements. You want to assign a different job to each student, but only if the student meets the job's skill requirement. Design an algorithm to determine the maximum number of jobs that you can successfully assign, and state the time complexity of the algorithm.

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

To prove that this is optimal, let us introduce some notations. Let an assignment of a student to a job be given by a pair  $(s, j)$ , where  $s$  is the student, and  $j$  is the job. Let  $L(s)$  be the skill level of student  $s$ , and let  $R(j)$  be the skill requirement of job  $j$ .

Now, assume there is an alternative strategy that also produces an optimal assignment. We order the assignments produced by each strategy in increasing order of the student's skill level and consider the **first** violation of the greedy policy by the alternative strategy. There are two ways a violation could occur:

**Claim.** The greedy strategy assigns student  $s_1$  the job  $j_1$ , but the alternative strategy assigns the same student the job  $j_2$ .

Since the greedy strategy assigned student  $s_1$  the job  $j_1$ , we have  $L(s_1) \geq R(j_1)$ . Also, since the greedy strategy assigns each student the job with the highest skill requirement that they meet the requirements for (that hasn't already been assigned), we necessarily have  $R(j_1) \geq R(j_2)$ , otherwise, the greedy strategy would not have assigned  $j_1$  to  $s_1$ . If the alternative strategy assigned  $j_1$  to a different student, say  $s_2$ , this student must also meet the skill requirement for  $j_2$  (since  $R(j_1) \geq R(j_2)$ ). Hence, we can modify the assignment made by the alternative strategy to adhere to the greedy policy by swapping the assignments of the two students.

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

In this case, the alternative strategy **must** have assigned  $j_1$  to some student, otherwise the assignment would not be optimal, as we would be able to add the assignment  $(s_1, j_1)$ . Hence, suppose that the alternative strategy assigned  $j_1$  to a student  $s_2$ . Since the greedy strategy considers students in increasing order of skill level, we necessarily have  $L(s_1) \leq L(s_2)$ . (If  $L(s_1) \geq L(s_2)$  then the greedy strategy would have assigned  $s_2$  a job first) But since the greedy strategy assigned  $j_1$  to  $s_1$ ,  $L(s_1) \geq R(j_1)$ . Hence, we can modify the assignment made by the alternative strategy to adhere to the greedy policy by assigning  $j_1$  to  $s_1$  rather than assigning it to  $s_2$ .

Hence, we have shown that for each possible violation, a modification can be made to the assignment to make it adhere to the greedy policy. Hence, if an assignment contains multiple violations, we can apply these modifications one by one, transforming the assignment into one that would be produced by the greedy strategy. Thus, any optimal assignment can be transformed into an assignment that adheres to the greedy policy, and therefore the greedy strategy is optimal.

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

**[K] Exercise 18.** You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if a job is accepted, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the activity selection problem.)

- (a) Given a list of  $n$  such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Design an  $O(n^2)$  algorithm to do this with a running time that is polynomial in  $n$ . You may assume for simplicity that no two jobs have the same start or end times.
- (b) Suppose that now for each of the  $n$  jobs given, an inspector is required to check the operation of the processor at any point during its time of operation at least once, however, you do not know which subset of the jobs will be chosen. Therefore, design an algorithm that finds the minimal and valid list of time stamps for the inspector to come and check the operation. You may assume that each inspection can be done instantaneously.

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

- (a) This part is similar to the *Activity Selection problem*, except that time is now a 24hr circle, rather than an interval because there might be jobs whose start time is before midnight and finishing time after midnight. However, we can reduce it to several instances of the interval case.

Let  $S = \{J_1, J_2, \dots, J_k\}$  be the set of all jobs whose start time is before midnight and finishing time is after midnight. We now first exclude all of these jobs and sort the remaining jobs by their finishing time. We now solve in the usual manner the activity selection problem with the remaining jobs only, by always picking a non-conflicting job with the earliest finishing time. We record the number of accepted jobs obtained in this manner as  $a_0$ . We now start over, but this time we take  $J_1$  as our first job, and we record the number of accepted jobs obtained as  $a_1$ . We repeat this process with the rest of the jobs in  $S$ , recording the number of accepted jobs as  $a_2, \dots, a_k$ . Finally, we pick the selection of jobs for which the corresponding  $a_m$  is the largest,  $0 \leq m \leq k$ .

To prove optimality, we note that the optimal solution either does not contain any of the jobs from the set  $S$  or contains just one of them. If it does not contain any of the jobs from  $S$ , then it would have been constructed when the problem was solved for all jobs excluding  $S$ , by the same argument as was given for the Activity Selection problem; if it does contain a job  $J_m$  from  $S$ , then it would have been constructed during the round when we started with  $J_m$ .

Sorting all jobs which are not in  $S = \{J_1, J_2, \dots, J_k\}$  takes at most  $O(n \log n)$  time. Each of the  $k + 1$  procedures runs in linear time (because we go through the list of all jobs by their finishing time only once, checking if their start time is before or after the finishing time of the last chosen activity). The number of rounds is at most  $n$ , so the time complexity is  $O(n \log n + n^2) = O(n^2)$ .

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

We start by picking the timestamp right at the finish time of  $J_1$ . Then, we consider the jobs that have not been stabbed and pick the job whose starting time occurs the earliest (with respect to the last picked timestamp) and stab it at its right endpoint. Repeat in this manner until all jobs have been stabbed, and then record the number of needles used as  $n_1$ . Now start over, but this time begin by stabbing  $J_2$  at its right endpoint, and then continue

in the above manner while wrapping around the end of the line and recording the number of needles used as  $n_2$ . Do this for each  $J_i$ , and then take the arrangement for which the number of needles used is minimal for the final solution.

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## § SECTION FOUR: GAMES AND GRAPHS

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

- (a) Assume that  $T$  is the minimal spanning tree of  $G$ . Does  $T$  necessarily remain the minimal spanning tree for the new graph  $G'$ ?
- (b) Assume that  $p$  is the shortest path from a vertex  $u$  to a vertex  $v$  in  $G$ . Does  $p$  necessarily remain the shortest path from  $u$  to  $v$  in the new graph  $G'$ ?

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

- (a) By considering how Kruskal's algorithm functions, it sorts the edges of the graph by weight (in ascending order) and then greedily selects and joins them until an MST is formed. Therefore, as long as the order of the edges is consistent between  $G$  and  $G'$ , Kruskal will produce the same result. This is precisely the case with  $w(i, j)^2$  as it is monotone over  $\mathbb{N}$ .
- (b) This is not the case, consider for example a graph  $G$  with vertices  $A, B, C$  and  $D$  and edges  $AB = 1, BD = 5, AC = 3, CD = 4, AD = 7$  and  $BC = 8$ . Then the shortest path from  $A$  to  $D$  in  $G$  is  $ABD$  with length 6, while the path  $ACD$  is longer, with length 7. However, after squaring all the weights, the length of  $ABD$  becomes  $1 + 25 = 26$ , while the length of  $ACD$  becomes  $9 + 16 = 25$  which is shorter. In a more abstract notion, there is **no guarantee** that for 2 paths with respective weights of  $\{w_1^{(1)}, w_2^{(1)}, \dots, w_k^{(1)}\}$  and  $\{w_1^{(2)}, w_2^{(2)}, \dots, w_n^{(2)}\}$ , we have that

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

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

□

**[K] Exercise 22.** Consider a complete binary tree with  $n = 2^k$  leaves. Each edge has an associated positive number that we call the length of the edge (see figure below). The distance from the root to a leaf is the sum of the lengths of all edges from the root to the leaf. The root emits a clock signal and the signal propagates along all edges and reaches each leaf in time proportional to the distance from the root to that leaf. Design an  $O(n)$  algorithm which increases the lengths of some of the edges in the tree in a way that ensures that the signal reaches all the leaves at the same time while the total sum of the lengths of all edges is minimal.

For example, in the picture below if the tree A is transformed into trees B and C all leaves of B and C have a distance of 5 from the root and thus receive the clock signal at the same time, but the sum of the lengths of the edges in C is 17 while sum of the lengths of the edges in B is only 15.

Problem Set 3/Section 4/VLSI1.pdf

*Solution.* We proceed recursively as follows:

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

To understand the algorithm, consider the following example.

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

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

Problem Set 3/Section 4/VLSI2.pdf

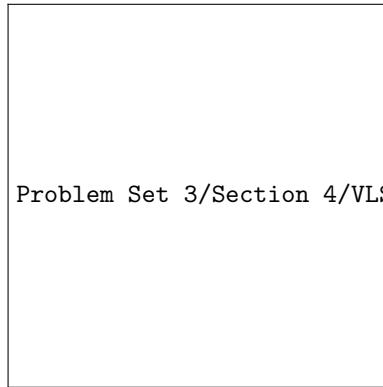
**Claim.** The signal reaches all of the leaves at the same time.

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

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

The algorithm then updates the smaller edge weight as follows:

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



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

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

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

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

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

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

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

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

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

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

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

- If  $x < y$ , then algorithm  $O$  chooses to increase any of the branches by a value smaller than the maximum of the two branches of concern. However, this yields an infeasible solution since either one of the branches

need to reduce in length or the subsequent signal can't reach the leaves at the same time. Hence, this case yields an invalid solution.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**[K] Exercise 26.** Define the Fibonacci numbers as

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

- (a) Design an algorithm to express a given positive integer  $n$  as a sum of non-consecutive Fibonacci numbers.
  - (b) Justify the correctness of your algorithm.
- You will need to argue that your algorithm finds Fibonacci numbers that sum to  $n$  and each of the Fibonacci numbers are non-consecutive.
- (c) Show that the solution obtained in part (a) is unique.

*Solution.*

- (a) Find the largest Fibonacci number less than or equal to  $n$ , and subtract it from  $n$ . Repeat until there is no remainder.
- (b) Let  $F_k$  be the largest Fibonacci number less than or equal to  $n$ . Then  $n < F_{k+1} = F_k + F_{k-1}$ , and therefore  $n - F_k < F_{k-1}$ . It follows that the largest Fibonacci number less than or equal to the remainder is at most  $F_{k-2}$ , which is not consecutive with  $F_k$  in the Fibonacci sequence.
- (c) Suppose for a contradiction that for some positive integer  $n$  such that  $F_k \leq n < F_{k+1}$ ,  $n$  can be written as a sum of non-consecutive Fibonacci numbers without using  $F_k$ . The sum is then at most  $F_{k-1} + F_{k-3} + F_{k-5} + \dots$ .

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

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

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

as required.

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

**[H] Exercise 27.** There are  $n$  stepping stones and each stepping stone has an associated positive integer, which represents the maximum number of stones you can skip past. You initially start at the first stone.

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

- (b) You now want to arrive at the last stone with the smallest number of jumps. Construct a greedy algorithm to obtain the sequence of jumps that minimises the number of jumps, clearly outlining what the greedy heuristic is.
- (c) Justify the correctness of your algorithm, using the “greedy stays ahead” method. To assist with your argument, you may want to use the following scaffold to ensure that your argument is on the right track.
- The inductive proof is based on the number of jumps. You should clearly outline why the base case (one stepping stone) follows from your greedy algorithm.
  - Clearly define the output of your greedy algorithm and the output of an arbitrarily chosen solution. Suppose that your greedy algorithm is at least *as far* as the chosen solution at jump  $k$ .
  - Show that, in jump  $k + 1$ , your greedy algorithm is also at least *as far* as the chosen solution.
  - Conclude that your greedy algorithm is as good as any arbitrary solution, which justifies the correctness of your algorithm.

*Solution.*

- (a) At each stone, we can always step to the next adjacent stone since each stepping stone has value  $\geq 1$ . This shows that it is always possible to reach the last stone.
- (b) The intuition is that we never want to miss out on a potential stone that gives greater distance. Therefore, a greedy heuristic of jumping as far as you can doesn’t work because you can miss out on stones which can give you more options in the long run. Come up with a counterexample to show that this is indeed a suboptimal approach in the general case.

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

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

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

□

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

- (a) Consider the following greedy algorithm:

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

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

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

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

*Solution.*

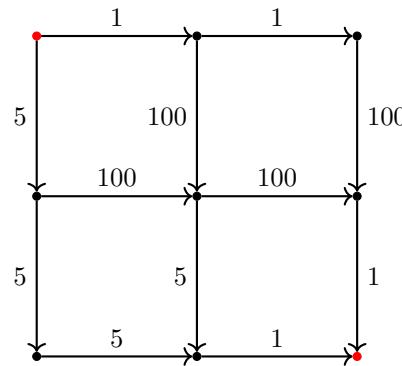
- (a) Consider the following  $3 \times 3$  grid.

|   |     |     |
|---|-----|-----|
| 1 | 1   | 1   |
| 5 | 100 | 100 |
| 5 | 5   | 1   |

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

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

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



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

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

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

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

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

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

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

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

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

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

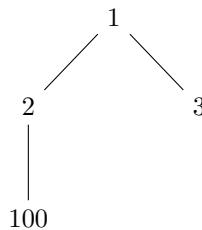
However, colouring a vertex has a cost, measured by when it is coloured and its associated weight. In other words, the  $k$ th vertex – say vertex  $a$  – that is coloured has cost  $k \times w_a$ . For example, the cost of colouring the root node  $r$  is  $1 \times w_r$  since the root of the tree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

- A basic greedy approach to this problem is to focus on the current vertex and colour all its children in descending order of weight. Show that this doesn't always yield a correct solution by providing a counterexample.
- Design an  $O(n^3)$  algorithm that finds the minimum total cost of colouring the whole tree. Justify the correctness of your algorithm.
- (Extended students only) Design an algorithm that solves the previous problem in  $O(n \log n)$ .

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

*Solution.*

- (a) Consider the following counter example.



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

the minimum total cost.

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

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

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

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

- $a_1, \dots, a_p, b_1, \dots, b_q$  with total cost:

$$C_1 = \sum_{i=1}^p ia_i + \sum_{i=1}^q (i+p)b_i;$$

- $b_1, \dots, b_q, a_1, \dots, a_p$  with total cost:

$$C_2 = \sum_{i=1}^q ib_i + \sum_{i=1}^p (i+q)a_i.$$

Then ordering 1 is better if

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

### Multiway Cut

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

**Task:** Return the size of the smallest multiway cut.

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

**[E] Exercise 33.**

# COMP3121/9101

## ALGORITHM DESIGN

### PROBLEM SET 4 – MAXIMUM FLOW

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

## Contents

|  |           |
|--|-----------|
| <b>1 SECTION ONE: MAXIMUM FLOW</b>         | <b>2</b>  |
| <b>2 SECTION TWO: MINIMUM CUT</b>          | <b>6</b>  |
| <b>3 SECTION THREE: BIPARTITE MATCHING</b> | <b>10</b> |

## § SECTION ONE: MAXIMUM FLOW

**[K] Exercise 1.** Several families are coming to a birthday celebration in a restaurant. You have arranged that  $v$  many tables will serve only vegetarian dishes,  $p$  many tables will not serve pork and  $r$  many remaining tables will serve food with pork. You know that  $V$  many families are all vegetarians,  $P_1$  many families do not eat pork but do not mind eating vegetarian dishes,  $P_2$  many families do not eat pork but hate vegetarian dishes. Also  $R_1$  many families have no dietary restrictions and would also not mind eating vegetarian dishes or food without pork,  $R_2$  many families have no dietary restrictions but hate vegetarian dishes but can eat food without pork. Finally,  $S$  many families are from Serbia and cannot imagine not eating pork. You are also given the number of family members in each family and the number of seats at each table.

In total, there are  $m$  families and  $n$  tables. You must place the guests at the tables so that their food preferences are respected and no two members from the same family sit at the same table. Your algorithm must run in time polynomial in  $m$  and  $n$ , and in case the problem has no solutions, your algorithm should output “no solution”.

*Solution.* We begin by constructing a bipartite flow network as follows:

- the left hand side vertices represent families,
- the right hand side vertices represent tables,
- source  $s$  and sink  $t$ ,
- connect  $s$  to each family vertex with capacity equal to the number of family members,
- connect each table vertex to  $t$  with capacity equal to the number of seats at that table,
- connect each family vertex with all tables compatible with the dietary preference with capacity 1.
  - The  $V$  vegetarian families are only connected to the  $v$  vegetarian tables.
  - The  $P_1$  non-pork families are connected to the  $v$  vegetarian tables and the  $p$  non-pork tables.
  - The  $P_2$  non-pork non-veg families are connected to the  $p$  non-pork tables only.
  - The  $R_1$  families who eat anything are connected to all tables.
  - The  $R_2$  non-veg families are connected to the  $p$  non-pork tables and the  $r$  pork tables.
  - The  $S$  Serbian tables are connected to the  $r$  pork tables only.
  - The edge capacity of 1 ensures that no two members of the same family go to the same table.

From this flow network construction, we run Edmonds-Karp to find the maximum flow. If the maximum flow is less than the total number of guests, then we output “no solution”.

Otherwise, if the maximum flow equals the total number of guests, then a placement is possible. Further, we can deduce a placement of guests at tables by examining which of the (family,table) edges carry flow: if there is 1 unit of flow from family  $i$  to table  $j$  then we seat a member of family  $i$  at table  $j$ . Applying *flow conservation* at each table ensures that the table capacities are respected.

The time complexity is  $O(|V||E|^2)$  where  $V = m+n+2$  and  $E \leq m+n+mn$ , so the algorithm runs in time polynomial in  $m$  and  $n$  as required.  $\square$

**[K] Exercise 2.** A band of  $m$  criminals has infiltrated a secure building, which is structured as an  $n \times n$  square grid of rooms, each of which has a door on all of its sides. Thus,

- from an internal room, we can move to any of the four neighbouring rooms

- from a room on the side of the building (or edge room), we can move to three other rooms or leave the building
- from a corner room, we can move to two other rooms or leave the building

The criminals were able to shut down the building's security system before entering, but during their nefarious activities, the security system became operational again, so they decided to abort the mission and attempt to escape. The building has a sensor in each room, which becomes active when an intruder is detected, but only triggers the alarm if it is activated again. Thus, the criminals may be able to escape if they can all reach the outside of the building without any two of them passing through the same room.

Design an algorithm which runs in time polynomial in  $m$  and  $n$  and, given the  $m$  different rooms which the criminals occupy when the security system is reactivated, determines whether all  $m$  criminals can escape without triggering the alarm.

*Solution.* We start by creating the flow graph as follows:

- for each room, say with coordinates  $(i, j)$ :
  - construct a pair of vertices  $v_{i,j}^{\text{in}}$  and  $v_{i,j}^{\text{out}}$ , and
  - place an edge of capacity 1 from  $v_{i,j}^{\text{in}}$  to  $v_{i,j}^{\text{out}}$ , representing a ‘vertex capacity’ of 1 for room  $(i, j)$ .
- for each pair of neighbouring rooms  $(i, j)$  and  $(i', j')$ , place edges of capacity 1 from  $v_{i,j}^{\text{out}}$  to  $v_{i',j'}^{\text{in}}$  and from  $v_{i',j'}^{\text{out}}$  to  $v_{i,j}^{\text{in}}$ .
- add a super source  $s$ , and place an edge of capacity 1 from  $s$  to the in-vertex of each of the  $m$  initially occupied rooms.
- add a super sink  $t$ , and place an edge of capacity 1 from the out-vertex of each edge room to  $t$ .

Now we can find maximum flow  $f$  in such a network via Edmonds-Karp algorithm. Then the problem has a solution if  $f = m$ . Note that our approach is optimal such that every vertex and every edge can belong to only one path, which corresponds to unit flows in our constructed flow graph. For the constraint that all paths must be disjoint, our augmentation of the vertices will ensure that each square can only be occupied by 1 path hence all such paths we consider will be disjoint. Therefore, the maximum flow will indicate the maximum number of paths that satisfies the constraint and hence if  $f = m$  indicates if such a problem is possible.

In our flow graph  $V = 2n^2 + 2$ , and we will have:

- $n^2$  edges from an in-vertex to an out-vertex,
- $4n(n - 1)$  edges between adjacent rooms,
- $m$  edges from  $s$  and
- $4(n - 1)$  edges to  $t$ .

Then the overall complexity of our solution is

$$O(2(n^2 + 1) \times (n^2 + 4n(n - 1) + m + 4(n - 1))^2) = O(n^6m^2),$$

which is polynomial in  $m$  and  $n$ . □

**[H] Exercise 3.** You have been told of the wonder and beauty of a very famous painting. It is painted in the hyper-modern style, and so it is simply an  $n \times n$  grid of squares, with each square coloured either black or white.

You have never seen this picture for yourself but have been told some details of it by a friend. Your friend has told you the value of  $n$  and the number of white squares in each row and each column. Additionally, your friend has also been

kind enough to tell you the specific colour of some squares: some squares are black, some are white, and the rest they simply could not remember.

The more details they tell you, the more amazing this painting becomes but you begin to wonder that perhaps it's simply too good to be true. Thus, you wish to design an algorithm which runs in time polynomial in  $n$  and determines whether or not such a painting can exist.

*Solution.* This problem can be viewed as a (bipartite) network flow problem in disguise. We begin by adjusting the count of the number of white squares in each row and column based on the location of the known (white) squares.

**Note.** The sum of the count in all rows must equal the sum of the count in all columns.

Let this sum be  $S$  so we can infer that there are precisely  $S$  white squares among the squares of unknown colour. We then consider the bipartite graph where every row is a vertex on the left side of the graph and every column is a vertex on the right side, making  $2n$  vertices in total. Every square in the grid which is of unknown colour forms a directed edge from its corresponding row to its corresponding column.

We can then convert this graph into a flow network as follows:

- each edge has capacity of 1.
- we add a source  $s$  and sink  $t$  to this graph.
- we add a directed edge from  $s$  to each row with capacity equal to the adjusted number of white squares in that row
- we add a directed edge from each column to  $t$  with capacity equal to the adjusted number of white squares in that column

It is evident that the saturated edges of the bipartite graph in any integer-valued flow from the source to the sink describe a possible colouring of the grid. Therefore we can now find the maximum flow  $f$  via the Edmonds-Karp algorithm and as any such flow has a capacity at most  $S$ , a painting exists if and only if  $f = S$ .

With  $2n$  vertices in total,  $2n$  edges for  $s$  and  $t$  and  $n^2$  edges for the bipartite flow graph, we can conclude that our solution runs in a time complexity of  $O(n^5)$ , which is clearly polynomial in  $n$ .  $\square$

**[H] Exercise 4.** Alice is the manager of a café which supplies  $n$  different kinds of drink and  $m$  different kinds of dessert.

One day the materials are in short supply, so she can only make  $a_i$  cups of each drink type  $i$  and  $b_j$  servings of each dessert type  $j$ .

On this day,  $k$  customers come to the café and the  $i$ th of them has  $p_i$  favourite drinks ( $c_{i,1}, c_{i,2}, \dots, c_{i,p_i}$ ) and  $q_i$  favourite desserts ( $d_{i,1}, d_{i,2}, \dots, d_{i,q_i}$ ). Each customer wants to order one cup of any one of their favourite drinks and one serving of any one of their favourite desserts. If Alice refuses to serve them, or if all their favourite drinks or all their favourite desserts are unavailable, the customer will instead leave the café and provide a poor rating.

Alice wants to save the restaurant's rating. From her extensive experience with these  $k$  customers, she has listed out the favourite drinks and desserts of each customer, and she wants your help to decide which customers' orders should be fulfilled.

Design an algorithm which runs in time polynomial in  $n, m$  and  $k$  and determines the smallest possible number of poor ratings that Alice can receive, given that:

- all  $p_i$  and all  $q_i$  are 1 (i.e. each customer has only one favourite drink and one favourite dessert),
- there is no restriction on the  $p_i$  and  $q_i$ .

*Solution.* (a) Construct a flow graph with a vertex  $A_i$  for each drink, a vertex  $B_j$  for each dessert, then two extra vertices for a source  $S$  and a sink  $T$ . For each drink  $i$ , add an edge with capacity  $a_i$  from  $S$  to  $A_i$ . For each dessert  $j$ , add an edge with capacity  $b_j$  from  $B_j$  to  $T$ . Finally, for each customer, add an edge of capacity 1 from  $c_{i,1}$  to  $d_{i,1}$ . The answer is  $k$  minus the maximum flow, found using Edmonds-Karp. The time complexity is  $O(VE^2)$ , where  $V = 2 + n + m$  and  $E = n + m + k$ , so it is polynomial in  $n, m$  and  $k$ .

(b) We start by constructing a flow graph with:

- Two vertices,  $S$  and  $T$ , for the source and the sink.
- A vertex  $A_i$  for each drink  $i$ , with an edge of capacity  $a_i$  from  $S$  to  $A_i$ , to restrict the number of available cups of this drink.
- A vertex  $B_j$  for each dessert  $j$ , with an edge of capacity  $b_j$  from  $B_j$  to  $T$ , to restrict the number of available servings of this dessert.
- Two vertices  $C_i$  and  $D_i$  for each customer, with an edge of capacity 1 from  $C_i$  to  $D_i$  to ensure that each customer either has both their drink and dessert, or has neither. Note that we ignore serving them only one, as that is equivalent to serving them nothing in terms of ratings.
- For each favourite drink  $c_{i,j}$  of customer  $i$ , an edge of capacity 1 from  $A_{c_{i,j}}$  to  $C_i$  for any drink they would accept.
- For each favourite dessert  $d_{i,j}$  of customer  $i$ , an edge of capacity 1 from  $D_i$  to  $B_{c_{i,j}}$  for any dessert they would accept.

Each unit of flow through this graph assigns a different customer one of their favourite drinks and one of their favourite desserts. Running the Edmonds-Karp algorithm on this graph then gives us the maximum flow, i.e. the maximum number of customers that we can satisfy, and so  $k$  minus this value is the minimum number of poor ratings.

Our flow graph has  $V = 2 + n + m + 2k$  vertices and  $E = n + m + k + \sum_{i=1}^k (p_i + q_i) \leq n + m + k + (n + m)k$  edges, so the time complexity of  $O(VE^2)$  is indeed polynomial in  $n, m$  and  $k$ .

□

## § SECTION TWO: MINIMUM CUT

**[K] Exercise 5.** There are  $n$  cities (labelled  $1, 2, \dots, n$ ), connected by  $m$  bidirectional roads. Each road connects two different cities. A pair of cities may be connected by multiple roads. A well-known criminal is currently in city 1 and wishes to get to the city  $n$  via road. To catch them, the police have decided to block the minimum number of roads possible to make it impossible to get from city 1 to city  $n$ . However, some roads are major roads. In order to avoid disruption, the police cannot close any major roads.

Your goal is to find the minimum number of roads to block to prevent the criminal from going from city 1 to city  $n$ , or report that the police cannot stop the criminal. Design an algorithm which achieves this goal and runs in time polynomial in  $n$  and  $m$ .

*Solution.* We construct a flow network as follows:

- create cities as vertices  $v_1, v_2, \dots, v_n$
- make  $v_1$  as the source and  $v_n$  as the sink
- suppose there are  $k$  roads between two cities  $i$  and  $j$ ,
  - if none of the roads is a major road, we connect  $v_i$  and  $v_j$  with two directed edges in opposite directions and of capacity each equal to  $k$ .
  - if one of the roads is a major road the capacity of the two directed edges is set to  $m + 1$ .

We can now find the maximum flow  $f$  in such a network using the Edmonds-Karp algorithm, and recall that the value of this flow equals the capacity of the min cut.

- If  $|f| > m$  then at least one edge of capacity  $m + 1$  has been crossed, which indicates that every cut is crossed by a major road which cannot be blocked and thus we cannot catch the criminal.
- On the other hand, if  $|f| \leq m$ , then there is a cut which is crossed only by minor roads, so the criminal can be caught. To block the fewest number of roads, we block those roads which cross the min cut in the forward direction, i.e. those which go from a vertex reachable from  $v_1$  in the final residual graph to a vertex without this property.

With a total of  $n$  nodes and a maximum of  $m$  edges connecting each of possible pairs of cities, the complexity of our algorithm is then  $O(nm^2)$ . □

**[K] Exercise 6.** In the country of Pipelistan there are several oil wells, several oil refineries and many distribution hubs all connected by oil pipelines. To visualise Pipelistan's oil infrastructure, just imagine a undirected graph with  $k$  source vertices (the oil wells),  $m$  sinks (refineries) and  $n$  vertices which are distribution hubs linking (unidirectional) pipelines incoming to this vertex with the outgoing pipelines from that vertex.

You are given the graph and the capacity  $C(i, j)$  of each pipeline joining a vertex  $i$  with vertex  $j$ . You want to install the smallest possible number of flow meters on some of these pipelines so that the total throughput of oil from all the wells to all refineries can be computed exactly from the readings of all of these meters. Each meter shows the direction of the flow and the quantity of flow per minute. Design an algorithm which runs in time polynomial in  $k, m$  and  $n$  and decides on which pipelines to place the flow meters.

*Solution.* We construct a flow network as follows:

- source  $s$  and sink  $t$ ,
- a vertex  $w_i$  for each oil well;
- for every  $i$ , we connect  $s$  to  $w_i$  of infinite edge capacity.

- a vertex  $r_j$  for each oil refinery;
  - for every  $j$ , we connect  $r_j$  to  $t$  of infinite edge capacity.
- a vertex  $d_\ell$  for each distribution hub,
- each pipeline is represented by two directed edges of opposite direction, which represents bidirectional edges, of edge capacities 1 each.

**Note.** Note that we've ignored the *actual* capacity of each pipeline.

Once the flow network is constructed, we run Edmonds-Karp to find the maximal flow. We finally construct the last residual network flow, and look at all vertices to which there is a path from the source  $s$ . This defines a minimum cut, so we look at all edges crossing the minimum cut. The number of such edges (in the forward direction only!) determines the minimal number of metres needed to accurately compute the total flow from  $s$  to  $t$ .

The time complexity of our algorithm is given by the time complexity of Edmonds-Karp, which is given by  $O(V \cdot E^2)$ . The number of vertices given in our network is maximally given by  $k+m+n+2$  ( $k$  oil wells,  $m$  refineries,  $n$  distribution hubs, as well as the source and sink). Maximally, we have  $k$  edges from the source to  $w_i$ ,  $m$  edges from  $r_j$  to the sink, and  $((k+m+n)(k+m+n-1))/2$  pipelines. To see why, note that we have  $k+m+n$  vertices which excludes the source and sink. We can have edges between refineries, distribution hubs, or oil wells. Hence, the maximal number of edges is given by  $\binom{k+m+n}{2}$ . This, the overall time complexity is given by  $O(V \cdot E^2)$ , where  $V = k+m+n+2$  and  $E = (k+m+n)^2/2 - (k+m+n)/2$ .  $\square$

**[K] Exercise 7.** Assume that you are given a network flow graph with  $n$  vertices, including a source  $s$ , a sink  $t$  and two other distinct vertices  $u$  and  $v$ , and  $m$  edges. Design an algorithm which runs in time polynomial in  $n$  and  $m$  and returns the smallest capacity-cut among all cuts for which the vertex  $u$  is on the same side of the cut as the source  $s$  and vertex  $v$  is on the same side as the sink  $t$ .

*Solution.* Take the given flow graph, we construct via

- create the source  $s$  connect the vertex  $u$  with an edge of  $\infty$  capacity
- create the sink  $t$  and connect vertex  $v$  with an edge of  $\infty$  capacity

We then use Edmonds-Karp algorithm to find the maximum flow through such a network and then the corresponding minimal cut. The two added edges of infinite capacity cannot belong to the min-cut which ensures that  $u$  stays at the same side as  $s$  and  $v$  at the same side as  $t$ .

The total complexity of our algorithm is  $O(nm^2)$ .  $\square$

**[K] Exercise 8.** Assume that you are given a network flow graph with  $n$  vertices, including a source  $s$ , a sink  $t$  and two other distinct vertices  $u$  and  $v$ , and  $m$  edges. Design an algorithm which returns a smallest capacity cut among all cuts for which vertices  $u$  and  $v$  are in the same side of the cut.

*Solution.* Given the flow network, we construct two directed edges, one from  $u$  to  $v$  and the other from  $v$  to  $u$ , both of which having infinite edge capacities respectively. We note that, if only one infinite edge is constructed from  $u$  to  $v$ , then  $v$  can still end up on the source side and  $u$  can still end up on the sink side, so the edge will not belong in the cut. Once the flow network is constructed, we then run Edmonds-Karp to find the maximum flow and its corresponding minimal cut. From the discussion above, the two directed edges will ensure that  $u$  and  $v$  remain in the same side of the cut. The time complexity is again  $O(nm^2)$ .  $\square$

**[K] Exercise 9.** Given an undirected graph with vertices numbered  $1, 2, \dots, n$  and  $m$  edges, design an algorithm which runs in time polynomial in  $n$  and  $m$  and partitions the vertices into two disjoint subsets such that:

- vertex 1 and  $n$  are in different subsets, and
- the number of edges with both ends in the same subset is maximised.

*Solution.* We take the given undirected graph and construct a flow network by

- designating vertex 1 as the source and vertex  $n$  as the sink, and
- replacing every undirected edge with 2 directed edges of capacity 1.

Our problem is now equivalent to minimising the number of edges between the two subsets. Note that all the original undirected edges appear in pairs of edges with the same endpoints but in opposite directions. Also, whenever there is an edge from the sink side to the source side, there will also be an edge in the opposite direction which the min-cut will take into account.

Therefore we can then simply apply the Edmonds-Karp algorithm to find the maximum flow through such a network and then the corresponding minimal cut. The total complexity of our algorithm is  $O(nm)$  as the flow  $f \leq m$ .  $\square$

**[K] Exercise 10.** You know that  $n + 2$  spies  $S, s_1, s_2, \dots, s_n$  and  $T$  are communicating through  $m$  communication channels; in fact, for each  $i$  and each  $j$  you know if there is a channel through which spy  $s_i$  can send a secret message to spy  $s_j$  or if there is no such a channel (i.e., you know what the graph with spies as vertices and communication channels as edges looks like). Design an algorithm which runs in time polynomial in  $n$  and  $m$  that prevents spy  $S$  from sending a message to spy  $T$  by:

- (a) compromising as few channels as possible;
- (b) bribing as few of the other spies as possible.

*Solution.* We proceed as below:

- (a) We construct a flow network with:

- each vertex  $v_i$  representing each spy  $i$
- edges of capacity 1 from  $v_i$  to  $v_j$  if there is a communication link from  $i$  to  $j$

We then run Edmonds-Karp on the flow network to find the maximum flow and the corresponding minimum cut. As any cut of the graph represents a valid set of channels (or edges) for our problem, then the edges that cross the minimum cut *forwards* are the ones that have to be compromised.

This runs in polynomial time as we have  $E = m$  and  $f \leq m$ , the total complexity of our algorithm is then  $O(E|f|) = O(m^2)$  time.

- (b) We approach this very similarly; we replicate the graph via part (a), then

- change all capacities of the edges to  $\infty$
- for each non-source and non-sink vertex  $v_i$ , we split  $v_i$  into two vertices  $v_i^{(\text{in})}$  and  $v_i^{(\text{out})}$ . For each edge incoming to  $v_i$ , we connect them to  $v_i^{(\text{in})}$ ; for each edge outgoing from  $v_i$ , we modify it to instead outgoing from  $v_i^{(\text{out})}$ . We lastly connect  $v_i^{(\text{in})}$  and  $v_i^{(\text{out})}$  with capacity of 1.

Using this flow network graph, we repeat the same process as part (a) by running Edmonds-Karp and computing the minimum cut to find all the edges that represent the corresponding spies to bribe. As we have  $E = m + n$  and  $f \leq n$ , our total complexity is slight different but still polynomial in  $m$  and  $n$  with  $O(E|f|) = n(m + n)$ .

□

## § SECTION THREE: BIPARTITE MATCHING

**[K] Exercise 11.** You are manufacturing integrated circuits from a rectangular silicon board that is divided into an  $m \times n$  grid of squares. Each integrated circuit requires two adjacent squares, either vertically or horizontally, that are cut out from this board. However, some squares in the silicon board are defective and cannot be used for integrated circuits. For each pair of coordinates  $(i, j)$ , you are given a boolean  $d_{i,j}$  representing whether the square in row  $i$  and column  $j$  is defective or not. Design an algorithm which runs in time polynomial in  $m$  and  $n$  and determines the maximum number of integrated circuits that can be cut out from this board.

*Solution.* We proceed by constructing a graph with:

- a vertex  $v_{i,j}$  for each non-defective cell, and
- an edge between each pair of neighbouring non-defective cells.

This graph is bipartite, as each edge joins a vertex where  $i + j$  is odd with a vertex where  $i + j$  is even (*think about why*).

Placing a circuit corresponds to selecting an edge in this graph. No cell can be part of more than one circuit, so no two edges can share a vertex, i.e. our selection of edges must constitute a matching. Therefore, the maximum number of circuits that can be placed is exactly the size of the maximum matching in this bipartite graph. This can be found using the Edmonds-Karp algorithm in  $O(VE)$  time (*why VE ?*), and since  $V \leq mn$  and  $E \leq 4mn$  the runtime is clearly polynomial in  $m$  and  $n$ .  $\square$

**[K] Exercise 12.** You are hosting a game festival where  $n$  players may participate in  $m$  games. During the festival event, each individual player has a preference of  $k$  candidates ( $k$  is the same for all players) – the  $i$ -th player has candidates  $a_{i1}, a_{i2}, \dots, a_{ik}$ . One player may choose at most one game (he may choose not to play, though) among those candidates.

For every game, the player having the highest score would receive a prize. It is guaranteed that no two players would share the same highest score (so you wouldn't have to worry about causing conflict between winners!). As the host of this game festival, you would like to know how many prizes should you prepare to ensure no player would end up receiving no prizes. That is, calculate the maximum number of distinct games chosen by players.

*Solution.* Construct the graph  $G = \langle V, E \rangle$  as follows:

- Define  $V_1$  to be the set of vertices that consist of the  $n$  players;
- Define  $V_2$  to be the set of vertices that consist of the  $m$  games;
- The graph will then consist of  $V = V_1 \cup V_2$  vertices, which is a disjoint union of  $V_1$  and  $V_2$ .
- For each of the  $k$  preferences, we draw an edge from player  $i \in V_1$  to game  $a_{ij} \in V_2$  to denote that player  $i$  likes game  $a_{ij}$ .

From this construction, it is easy to see that such a graph  $G$  is *bipartite*. Therefore, the problem reduces to a maximum bipartite matching problem. Using the standard reduction to a maximum flow problem (construct a super source vertex  $s$  and a super sink vertex  $t$ ; construct an edge from  $s$  to every vertex in  $V_1$  and, for each vertex in  $V_2$ , construct an edge to  $t$ ; each edge in the new graph has capacity 1), run Edmonds-Karp to obtain the maximum flow. The maximum flow corresponds to the maximum matching of the original problem.  $\square$

**[K] Exercise 13.** You are given an  $n \times n$  chessboard. with  $k$  white bishops on the board at the given cells  $(a_i, b_i)$ , where  $1 \leq a_i, b_i \leq n$  for each  $1 \leq i \leq k$ . You have to determine the largest number of black rooks which you can place

on the board so that no two rooks are in the same row or in the same column or are under the attack of any of the  $k$  bishops. Recall that bishops attack *diagonally*.

*Solution.* To solve this problem we construct a bipartite graph with  $n$  left vertices  $r_i$  representing  $n$  rows of the board and  $n$  right vertices  $c_j$  representing  $n$  columns of the board. We construct edges in such a graph so that vertex  $r_i$  is connected with a vertex  $c_j$  just in case the cell  $(i, j)$  on the board is not under attack of any of the bishops. We add a super source  $s$  and connect it with all vertices  $r_i$  with edges of capacity 1; we also add a super sink and connect all vertices  $c_j$  also with edges of capacity 1. The maximal number of rooks that meet the conditions is equal to the max flow in this flow network, with rooks placed in the cells corresponding to the occupied edges from  $r_i$  to  $c_j$ .  $\square$

**[H] Exercise 14.** You are the head of  $n$  spies, who are all wandering in a city. On one day you received a secret message that the bad guys in this city are going to arrest all your spies, so you'll have to arrange for your spies to run away and hide in strongholds. You have  $T$  minutes before the bad guys arrive. Your  $n$  spies are currently located at

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n),$$

and your  $m$  strongholds are located at

$$(a_1, b_1), (a_2, b_2), \dots, (a_m, b_m).$$

The  $i$ th spy can move  $v_i$  units per minute, and each stronghold can hold only one spy.

Design an algorithm which runs in time polynomial in  $n$  and  $m$  determines which spies should be sent to which strongholds so that you have the maximum number of spies hiding from the bad guys.

*Solution.* First, for each spy  $i$  check which strongholds  $j$  are reachable in  $T$  minutes. Each of  $mn$  pairs can be checked in constant time by directly comparing the distance between spy  $i$  and stronghold  $j$  to  $v_i T$ , the distance that spy  $i$  can travel.

Then, we observe that matching spies with strongholds they can go to is a problem of finding the maximum bipartite matching, since no two spies can go to the same stronghold. We therefore construct a flow graph, with each spy  $i$  represented by a vertex  $p_i$  and each stronghold  $j$  represented by a vertex  $q_j$ , as well as a source  $s$  and a sink  $t$ . We place edges of capacity 1 from  $s$  to each  $p_i$ , and edges of capacity 1 from each  $q_j$  to  $t$ , and finally for each pair  $(i, j)$  such that spy  $i$  can reach stronghold  $j$ , we place an edge of capacity 1 from  $p_i$  to  $q_j$ . Running the Ford-Fulkerson algorithm on this graph finds a maximum flow, and hence the size of the maximum bipartite matching. We inspect the flow function found by this algorithm to recover the actual maximum matching – for each edge  $(p_i, q_j)$  carrying flow, we send spy  $i$  to stronghold  $j$ .

There are  $n + m$  total nodes and up to  $nm + n + m = O(nm)$  edges, so constructing the graph takes  $O(nm)$  time. Further, the value of a maximum flow is bounded above by  $\min(n, m)$ , so finding the maximum flow and hence the maximum matching takes  $O(nm \times \min(n, m))$  time.  $\square$

**[H] Exercise 15.** You have  $n$  warehouses and  $n$  shops. At each warehouse, a truck is loaded with enough goods to supply one shop. There are  $m$  roads, each going from a warehouse to a shop, and driving along the  $i$ -th road takes  $d_i$  hours, where  $d_i$  is an integer. Design a polynomial time algorithm to send the trucks to the shops, minimising the time until all shops are supplied.

*Hint:* Combine binary search with a max flow.

*Solution.* First, sort time travel distances  $d_i$  in increasing order; for the rest of the solution, we may now assume that  $d_{i+1} \geq d_i$ .

Consider a value  $d_i$  for some  $i$  and construct a bipartite graph  $G_i$  with warehouses  $w_j$  as the left side of the partition and with shops  $s_j$  ( $1 \leq j \leq n$ ). Connect all warehouses with all shops which are within travel distance times  $d_i$ . Use max flow to see if such bipartite graph has a perfect maximum matching of size  $n$ . Use a binary search to find the smallest  $i$  such that graph  $G_i$  has a matching of size  $n$ , i.e. a matching in which every warehouse has been matched with a shop, so that different warehouses are assigned different shops.  $\square$

**[H] Exercise 16.** There are  $n$  boys and  $n$  girls at a party. Whenever a song starts, they will form exactly  $n$  pairs to dance and no boy will dance with the same girl twice.

Some pairs of boys and girls like each other, and all other pairs of boys and girls dislike each other. Every boy will dance with at most  $k$  girls that he dislikes, and each girl will dance with at most  $k$  boys that she dislikes where  $k < n$ .

As the DJ, it is your job to determine the maximum number of songs to play such that it is possible for pairs to be formed that satisfy the above requirement. Design a  $O(n^4 \log n)$  algorithm that achieves this task.

*Hint: Start with the case where  $k = 0$  and fix a capacity  $x$  for edges between each boy and girl. How can you generalise this to arbitrary  $k$ ?*

*Solution.* We first take care of the case where  $k = 0$ . Label the boys  $b_1, b_2, \dots, b_n$ , and the girls  $g_1, g_2, \dots, g_n$ , and fix  $x \leq n$ . Construct a flow network with:

- a vertex  $b_i$  for each boy, and a vertex  $g_j$  for each girl
- vertices  $s$  and  $t$ , the super source and super sink
- for each boy, an edge from  $s$  to  $b_i$  with capacity  $x$
- for each girl, an edge from  $g_j$  to  $t$  with capacity  $x$
- for each boy-girl pair who like each other, an edge from  $b_i$  to  $g_j$  with capacity 1.

The total capacity leaving  $s$  (and entering  $t$ ) is  $nx$ , so the value of a maximum flow is at most  $nx$ . If  $x$  songs can be played, then we can construct a flow of value  $nx$  by flowing:

- all edges  $(s, b_i)$  with  $x$  units of flow
- all edges  $(g_j, t)$  with  $x$  units of flow
- for each pair  $(b_i, g_j)$  who dance together, the edge  $(b_i, g_j)$  with one unit of flow.

Proving the converse (that a flow of exactly  $nx$  allows  $x$  songs to be played) is more subtle.

Proof: Consider a maximum flow in the network. For each edge  $(b_i, g_j)$  which carries flow, we will record that boy  $i$  and girl  $j$  dance together for one song. If the maximum flow is  $nx$ , we will thus find  $x$  partners for each attendee, while enforcing that:

- no pair who dislike each other dances together, and
- no pair dances together more than once.

However, we have yet to confirm whether the  $nx$  boy-girl pairings can be grouped into  $x$  songs. The proof of this fact was not required to achieve full marks, but it is included here for completeness.

Construct a graph with  $2n$  vertices corresponding to the boys and girls, with  $nx$  edges corresponding to the matched pairs. This graph is clearly bipartite, with parts  $B = \{b_1, \dots, b_n\}$  and  $G = \{g_1, \dots, g_n\}$ , and  $x$ -regular (every vertex is incident to exactly  $x$  edges). Now, we use Hall's marriage theorem.

Definition: For a subset  $W$  of  $B$ , let  $N(W) \subseteq G$  be the set of vertices adjacent to at least one vertex in  $W$ .

Theorem\*: Suppose for all subsets  $W$  of  $B$  that  $|W| \leq |N(W)|$ . Then the graph has a perfect matching, i.e. a matching of size  $n$ .

For a contradiction, suppose there is a set  $W$  of  $p$  boys to which only  $q < p$  girls are adjacent. There are exactly  $px$  edges between  $W$  and  $N(W)$ , since each boy is matched with exactly  $x$  girls. However, each of these edges is also incident to exactly one of the  $q$  girls, and since  $p > q$  it is impossible for all  $q$  of these girls to have fewer than  $x$  edges each. Therefore, we have the required contradiction, and it follows that a perfect matching exists (corresponding to the pairs who dance together in the first song). Furthermore, upon removing these edges, the remaining graph is  $(x-1)$ -regular, so the same property applies. We can thus make  $n$  pairs for each of the  $x$  songs, as required.  $\square$

Thus, we have a test for whether  $x$  songs can be played or not. We can then apply this test for  $x = 0, 1, 2, \dots$  until we find the largest value of  $x$  for which it is possible, which is the answer.

We run the Edmonds-Karp algorithm up to  $n+1$  times. In each iteration, the number of edges is  $O(n^2)$  and the maximum flow is  $O(n^2)$ , so the total time complexity is  $O(n^5)$ . Note that this can be improved to  $O(n^4 \log n)$  by binary searching for the largest allowable value of  $x$ . This is clearly polynomial in  $n$  - running max flow on  $O(n)$  vertices,  $O(n^2)$  edges at most  $n$  times.

To generalise to arbitrary  $k$ , we consider the following construction.

The fundamental structure is as above: for each  $x$  from 0 to  $n$ , run the Edmonds-Karp algorithm once to determine whether  $x$  songs can be played, and return the largest  $x$  for which it was possible. However, the graph construction is more intricate. For a particular value  $x$ , construct a flow graph with:

- vertices  $b_i, b_i^L$  and  $b_i^D$  for each boy
- vertices  $g_j, g_j^L$  and  $g_j^D$  for each girl
- vertices  $s$  and  $t$ , the super source and super sink
- for each boy:
  - an edge from  $s$  to  $b_i$  with capacity  $x$
  - an edge from  $b_i$  to  $b_i^L$  with capacity  $x$
  - an edge from  $b_i$  to  $b_i^D$  with capacity  $k$
- for each girl:
  - an edge from  $g_j$  to  $t$  with capacity  $x$
  - an edge from  $g_j^L$  to  $g_j$  with capacity  $x$
  - an edge from  $g_j^D$  to  $g_j$  with capacity  $k$
- for each boy-girl pair:
  - an edge from  $b_i^L$  to  $g_j^L$  with capacity 1 if they like each other, or
  - an edge from  $b_i^D$  to  $g_j^D$  with capacity 1 if they don't like each other.

As above, we will find a maximum flow in this graph, and record each boy-girl edge carrying flow as a pair who dance together. This guarantees that:

- no pair dances together more than once, since for each pair  $(i, j)$ , either
  - they like each other, so  $c(b_i^L, g_j^L) = 1$  and  $c(b_i^D, g_j^D) = 0$ , or
  - they don't like each other, so  $c(b_i^L, g_j^L) = 0$  and  $c(b_i^D, g_j^D) = 1$ .
- each boy and each girl has exactly  $x$  partners
- of these partners, at most  $k$  are not liked.

The asymptotic time complexity is unaffected, since the number of edges is still  $O(n^2)$ .

Note that setting  $k = 0$  in this graph recovers the construction from before.  $\square$

# COMP3121/9101

## ALGORITHM DESIGN

### PRACTICE PROBLEM SET 5 – DYNAMIC PROGRAMMING

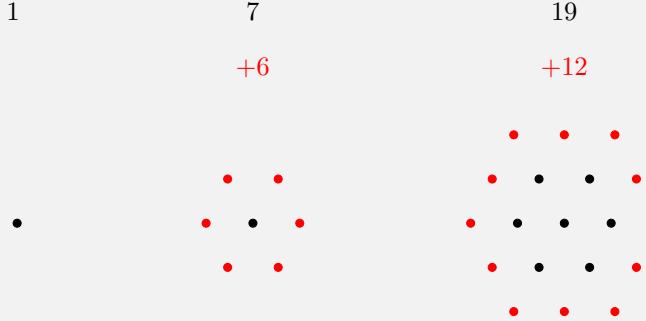
[K] – key questions [H] – harder questions [E] – extended questions [X] – beyond the scope of this course

## Contents

|                                  |    |
|----------------------------------|----|
| 1 SECTION ONE: MEMOISATION       | 2  |
| 2 SECTION TWO: OPTIMISATION      | 9  |
| 3 SECTION THREE: COUNTING        | 19 |
| 4 SECTION FOUR: GAMES AND GRAPHS | 23 |

## § SECTION ONE: MEMOISATION

[K] **Exercise 1.** The  $n$ th centred hexagonal number is the number of vertices required to fill a hexagon of radius  $n$ . The first few hexagonal numbers are given with an illustration.



Design an  $O(n)$  algorithm that returns the  $n$ th centred hexagonal number. Can you find a closed form for the  $n$ th centred hexagonal number, and thus, design an  $O(1)$  algorithm for the  $n$ th centred hexagonal number?

*Solution.* We define each subproblem  $P(i)$ : let  $\text{OPT}(i)$  be the  $i$ th centred hexagonal number.

Let  $H(n)$  be the  $n$ th centred hexagonal number. To get the  $n$ th layer of the hexagon, we need to add  $n$  vertices to each edge. Therefore, there are  $6n$  vertices we need to add. However, there are exactly 6 vertices that belong on two edges (these are the corner vertices of the hexagon). Therefore, we need to subtract the double counting of the corner vertices. In other words, our recursion becomes

$$H(n) = H(n - 1) + 6n - 6 = H(n - 1) + 6(n - 1),$$

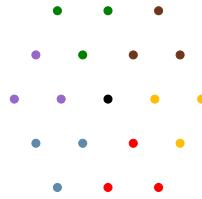
with  $H(1) = 1$ . This gives us a nice procedure for our recursive algorithm.

If  $n = 1$ , then our algorithm returns  $H(1) = 1$ . Otherwise, for each  $i$  from  $2, \dots, n$ , we return

$$H(i) = H(i - 1) + 6(i - 1).$$

We cache each intermediary subproblem into an array of length  $n$  so that the lookup time is  $O(1)$ .

We can actually partition the vertices (except for the centre) of the hexagon into six triangles as follows.



These form the  $(n - 1)$  triangular numbers which have the closed form

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2}.$$

Therefore, the  $n$ th centred hexagonal number can be thought of six  $(n - 1)$  triangular numbers with the centre vertex

which gives the closed form

$$\begin{aligned} H(n) &= 6 \left( \frac{n(n-1)}{2} \right) + 1 \\ &= 3n(n-1) + 1 \\ &= 3n^2 - 3n + 1. \end{aligned}$$

In other words, we can output the  $n$ th centred hexagonal number in constant time by returning  $H(n) = 3n^2 - 3n + 1$ .  $\square$

**[K] Exercise 2.** You are given a  $2 \times n$  rectangular board and identical  $1 \times 2$  tiles. Each tile can be laid either horizontally or vertically.

- (a) Design an  $O(n)$  algorithm to count the number of ways to completely cover the board with the  $1 \times 2$  tiles. Can you find a closed form?
- (b) We now build up a way to generate the number of ways to completely tile a  $3 \times n$  rectangular board. Let  $f(n)$  be the number of ways to tile a  $3 \times n$  board with  $1 \times 2$  tiles.
  - (i) Explain why  $f(2k+1) = 0$  for each  $k \in \mathbb{N}$ . For the remainder of this problem, we will let  $f(n)$  be the number of ways to tile a  $3 \times 2n$  board with  $1 \times 2$  tiles.
  - (ii) Show that  $f(1) = 3$  and  $f(2) = 11$ .
  - (iii) Generate a suitable recursion based on the tilings found in (i).

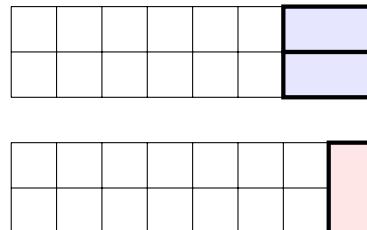
To ensure that your recursion is correct, check that  $f(2) = 11$ .

- (iv) Hence, design an  $O(n)$  algorithm that counts the number of ways to completely cover the  $3 \times 2n$  board with the tiles.

*As you found, constructing a general recurrence for an  $m \times n$  board is quite difficult.*

*Solution.*

- (a)
  - **Subproblem:** Let  $\text{NUM}(i)$  be the number of ways to tile a  $2 \times i$  rectangular board using  $1 \times 2$  tiles.
  - **Recurrence:** We observe that, to tile a  $2 \times i$  tile, we have the two ending possibilities as follows:



If we place one of the blue tiles, the other  $1 \times 2$  blue tile must appear to complete the grid. Therefore, in this case, it suffices to count the number of ways to tile the  $1 \times (i-2)$  board. If we decide to place a red tile, then we see that it suffices to count the number of ways to tile the  $1 \times (i-1)$  board. Therefore, we obtain the natural recurrence

$$\text{NUM}(i) = \text{NUM}(i-1) + \text{NUM}(i-2).$$

- **Base cases:** Since the recurrence relies on  $\text{NUM}(i - 1)$  and  $\text{NUM}(i - 2)$ , we require the two base cases:

$$\text{NUM}(1) = 1, \quad \text{NUM}(2) = 2.$$

- **Order of Computation and Final Solution:** By our recurrence, we observe that each subproblem relies on computing smaller sized boards. Therefore, the natural ordering is to solve the subproblems in increasing order of board size; that is, we compute  $\text{NUM}(1), \text{NUM}(2), \text{NUM}(3), \text{NUM}(4), \dots, \text{NUM}(n)$ , with the final solution being  $\text{NUM}(n)$ .
- **Time Complexity:** There are  $n$  subproblems, each of which can be computed in  $O(1)$  time. Therefore, the time complexity is  $O(n)$ .

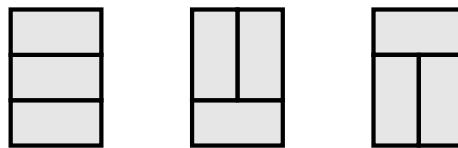
If we define  $\text{NUM}(0) = 1$ , then we observe that we obtain the Fibonacci sequence by our recurrence; specifically,

$$\begin{aligned} \text{NUM}(0) &= 1 = F_1, \\ \text{NUM}(1) &= 1 = F_2, \\ \text{NUM}(2) &= 2 = F_3, \\ \text{NUM}(3) &= 3 = F_4, \\ &\dots \\ \text{NUM}(n) &= F_{n-1}. \end{aligned}$$

Therefore, we can simply return  $\text{NUM}(n) = F_{n-1}$ , which has a well-known closed form; that is,

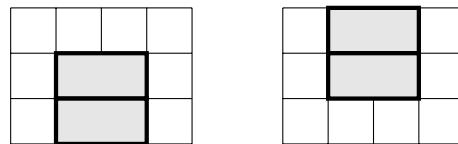
$$\text{NUM}(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}}.$$

- (b) (i) We see that every  $1 \times 2$  tile must cover exactly two distinct squares of the board. Therefore, to cover a board with  $1 \times 2$  tiles, the number of squares in the board must be even. However, any  $3 \times (2k + 1)$  board will contain an odd number of squares. Therefore, any tiling will have at least one square remaining if no tiles can intersect.
- (ii) We first show that  $f(1) = 3$ . This is easy to list out. We have

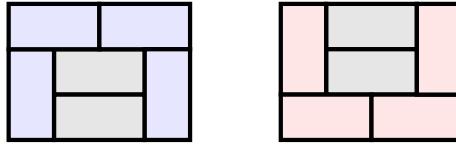


Therefore, we have  $f(1) = 3$ . To derive an expression for  $f(2)$ , we can break this into two  $3 \times 2$  boards. Each board has exactly three configurations as shown above. Therefore, there are  $3 \times 3 = 9$  configurations if tiles do not overlap between the two  $3 \times 2$  boards. We now look at all configurations where tiles can overlap over the two boards.

Consider the following configurations.



This forces the remaining tiles to be placed in the following positions.



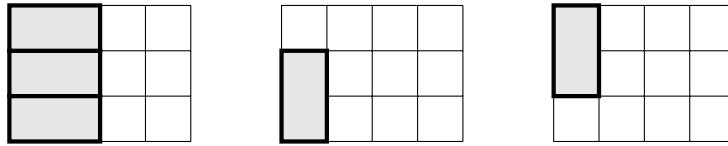
Note that placing three  $1 \times 2$  tiles in the middle of the board reduces the problem to a  $3 \times 1$  board which has no solution. Therefore, we note that these are the only additional solutions that aren't accounted for in the previous disjoint case. Thus, we obtain  $f(2) = 9 + 2 = 11$  number of ways to tile a  $3 \times 4$  board.

- (iii) Let  $f(n)$  denote the number of ways to tile a  $3 \times 2n$  board and  $g(n)$  denote the number of ways to tile a  $3 \times (2n+1)$  board with its first square missing.

To tile a  $3 \times 2n$  board, we can either:

- tile it so that the three left-most squares are all covered by the horizontal  $1 \times 2$  tiles, leaving us with a  $3 \times (2n-2)$  board,
- or tile two of the left-most squares leaving one square vacant.

The following are illustrated below.



This gives  $f(n) = f(n-1) + 2g(n-1)$ .

In a similar light, to tile a  $3 \times (2n+1)$  board with one tile missing, the other two squares have to be covered by either:

- a single tile, leaving a  $3 \times 2n$  board,
- or two tiles leaving a missing square in the next column.

This gives  $g(n) = f(n) + g(n-1)$ .

Therefore the recurrence becomes

$$\begin{aligned} f(n) &= f(n-1) + 2g(n-1) \\ &= f(n-1) + 2(f(n-1) + g(n-2)) \\ &= f(n-1) + 2\left(f(n-1) + \frac{f(n-1) - f(n-2)}{2}\right) \\ &= f(n-1) + 2f(n-1) + f(n-1) - f(n-2) \\ &= 4f(n-1) - f(n-2). \end{aligned}$$

- (c) We therefore devise a dynamic programming solution based on the recurrence above.

- **Subproblem:** Let  $f(n)$  denote the number of ways to tile a  $3 \times 2n$  board using  $1 \times 2$  dominoes.
- **Recurrence:** The recurrence is given by

$$f(n) = 4f(n-1) - f(n-2).$$

- **Base case:**  $f(1) = 3$  and  $f(2) = 11$ .
- **Order of Computation and Final Solution:** We solve each subproblem in increasing order of  $i$  with the final solution being  $f(n)$ .

- **Time Complexity:** There are  $n$  subproblems and each subproblem can be computed in constant time giving an  $O(n)$  algorithm.

□

**[K] Exercise 3.** In this problem, we introduce the *Tower of Hanoi* game. For students who might not be familiar with the game, we have three pegs and  $n$  disks. Each disk has a unique radius. In other words, no two distinct disks share the same radius. The rules of the game are as follows:

- The game state initially starts out with all disks on the first peg. The disks are stacked in increasing order of radius with the smallest disk on the top of the stack.
- In each move, you can only move one disk from one peg to another. You can only move one disk from one peg to another peg if the top of the stack of the peg has a bigger radius. In other words, you cannot stack a disk on top of a smaller disk.
- The game ends when all disks are placed in the third peg, in increasing order of radius with the smallest disk on the top of the stack of the third peg.

To familiarise yourself with the game, feel free to [click here](#) to play a game! In this problem, we will design an algorithm that finds the minimal number of moves to win the game, given  $n$  disks.

Let  $f(n)$  be the minimum number of moves to win the *Tower of Hanoi* game with  $n$  disks.

- Find  $f(1)$ ,  $f(2)$ ,  $f(3)$ , and  $f(4)$ .
- For  $n \geq 2$ , explain why  $f(n) = 2f(n - 1) + 1$ .
- Hence, design an  $O(n)$  algorithm that computes the minimum number of moves to win the game with  $n$  disks.

*Solution.*

- Denote  $a \rightarrow b$  as moving the topmost disk of peg  $a$  to peg  $b$ . Then:
  - for  $n = 1$ , the solution is  $1 \rightarrow 3$ ,
  - for  $n = 2$ , the solution is  $1 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 3$ ,
  - for  $n = 3$ , the solution is  $1 \rightarrow 3$ ,  $1 \rightarrow 2$ ,  $3 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 1$ ,  $2 \rightarrow 3$ ,  $1 \rightarrow 3$ , and
  - for  $n = 4$ , the solution is  $1 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 3$ ,  $1 \rightarrow 2$ ,  $3 \rightarrow 1$ ,  $3 \rightarrow 2$ ,  $1 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 3$ ,  $2 \rightarrow 1$ ,  $3 \rightarrow 1$ ,  $2 \rightarrow 3$ ,  $1 \rightarrow 2$ ,  $1 \rightarrow 3$ ,  $2 \rightarrow 3$ .

Therefore  $f(1) = 1$ ,  $f(2) = 3$ ,  $f(3) = 7$  and  $f(4) = 15$ .

- In order to move the largest disk to the target position, we must first move all  $n - 1$  disks above it to peg 2. This takes  $f(n - 1)$  moves. Once we move the largest disk, we must then move all  $n - 1$  smaller disks from peg 2 to peg 3, again taking  $f(n - 1)$  moves. Therefore the task both requires at least  $2f(n - 1) + 1$  moves and can be accomplished in exactly this many moves, completing the proof.
- Subproblems:** for  $i \geq 1$ , let  $f(i)$  be the minimum number of moves to win the *Tower of Hanoi* game with  $i$  disks.

**Recurrence:** for  $i > 1$ ,  $f(i) = 2f(i - 1) + 1$ .

**Base case:**  $f(1) = 1$ .

**Order of computation:** increasing from  $i = 1$  to  $i = n$ .

**Final answer:**  $f(n)$ .

**Time complexity:** each of  $n$  subproblems is solved in constant time, for a total time complexity of  $O(n)$ .

*Note:* we can also prove (by induction) that  $f(n) = 2^n - 1$ , and compute this directly for a faster algorithm.

□

**[H] Exercise 4.** Strings are formed from the alphabet  $\Sigma = \{0, 1, 2, 3\}$ . We want to count the number of strings of length  $n$  that do not contain 12 or 20 as substrings. Let  $f(n)$  be the number of such strings of length  $n$ .

(a) Find  $f(1)$  and  $f(2)$ .

(b) Show that  $f(3) = 49$ .

*Hint:* First, how many strings are there without any restriction? Then find the number of length 3 strings which contain 12 as a substring. Then find the number of length 3 strings which contain 20 as a substring. Are there any strings which contain *both*, 12 and 20 as substrings?

(c) Find a suitable recursion in terms of  $n - 1$ ,  $n - 2$ ,  $n - 3$  that finds the number of length  $n$  strings that do not contain 12 or 20 as substrings, and hence, design an  $O(n)$  algorithm that finds the number of  $n$  length strings that does not contain 12 or 20 as substrings.

Which subproblems would we need to consider to define our recursion?

*Solution.*

(a)  $f(1) = 4$ , as all single-character strings are valid.

$f(2) = 16 - 2 = 14$ , as all two-character strings except “12” and “20” are valid.

(b) There are four strings of each of the following forms: “12x”, “x12”, “20x” and “x20”. This suggests deducting sixteen from the set of sixty-four three character strings.

However “120” is double-counted, as it counts towards both the forms above. Therefore only fifteen three-character strings are invalid, giving the answer  $f(3) = 49$ .

(c) To obtain a suitable recurrence, let  $f(n - 1)$  denote the number of such strings of length  $n - 1$ . We now look at what happens when we append each of the characters. We will define four new recurrences:  $a(n)$ ,  $b(n)$ ,  $c(n)$ ,  $d(n)$  for such strings ending in 0, 1, 2, 3 respectively such that  $f(n) = a(n) + b(n) + c(n) + d(n)$ .

- The  $n$ th character is a 0. Then we cannot have the  $(n - 1)$ th character be a 2. Therefore, the  $(n - 1)$ th character is either a 0, 1 or 3. This gives the recurrence  $a(n) = a(n - 1) + b(n - 1) + d(n - 1)$ .
- The  $n$ th character is a 1. Then every string of length  $n - 1$  forms a valid string. Therefore, the number of such strings is  $b(n) = f(n - 1)$ .
- The  $n$ th character is a 2. Then we cannot have the  $(n - 1)$ th character be a 1. Therefore, the  $(n - 1)$ th character is either a 0, 2 or 3. This gives the recurrence  $c(n) = a(n - 1) + c(n - 1) + d(n - 1)$ .
- The  $n$ th character is a 3. Then every string of length  $n - 1$  forms a valid string. Therefore, the number of such strings is  $d(n) = f(n - 1)$ .

We now simplify this expression entirely in terms of  $f(n)$ . Note that  $b(n)$  and  $d(n)$  are recurrences purely in terms of  $f(n)$ . We also see that

$$f(n - 1) - b(n - 1) = a(n - 1) + c(n - 1) + d(n - 1) = c(n).$$

Therefore, we have that

$$\begin{aligned} c(n) &= f(n-1) - b(n-1) \\ &= f(n-1) - b(n-1) \\ &= f(n-1) - f(n-2). \end{aligned}$$

Similarly, we see that  $a(n) = f(n-1) - c(n-1)$  and

$$a(n) = f(n-1) - f(n-2) - f(n-3).$$

Thus, we deduce that the recurrence is

$$\begin{aligned} f(n) &= a(n) + b(n) + c(n) + d(n) \\ &= (f(n-1) - f(n-2) - f(n-3)) + f(n-1) + (f(n-1) - f(n-2)) + f(n-1) \\ &= 4f(n-1) - 2f(n-2) - f(n-3). \end{aligned}$$

We now design an  $O(n)$  algorithm based on the above.

- **Subproblem:** Let  $f(n)$  denote the number of strings of length  $n$  that do not contain 12 or 20 as substrings.
- **Recurrence:**  $f(n) = 4f(n-1) - 2f(n-2) - f(n-3)$  for  $n > 3$ .
- **Base case:**  $f(1) = 4$ ,  $f(2) = 14$ ,  $f(3) = 49$ .
- **Order of Computation and Final Solution:** We solve the subproblems in increasing order with the final solution being  $f(n)$ .
- **Time Complexity:** There are  $n$  subproblems and each subproblem can be computed in  $O(1)$  time; therefore, the running time is  $O(n)$ .

□

## § SECTION TWO: OPTIMISATION

**[K] Exercise 5.** (*Making Change*) You are given  $n$  types of denominations of values with  $v(1) < v(2) < \dots < v(n)$  and  $v(1) = 1$ . You are additionally given a positive integer  $C$ . Design an  $O(nC)$  algorithm which makes change of value  $C$  using as few coins as possible. You may assume that you have an infinite supply of such coins.

Provide a suitable subproblem that allows you to define a simple recursion.

*Solution.* To provide a formal dynamic programming solution, we need to propose a suitable subproblem for which our recursion falls out naturally.

Let  $P(i)$  be the subproblem: *what is the smallest number of coins required to make change of value  $i$ ?*. Let  $\text{OPT}(i)$  be the solution to the subproblem  $P(i)$ . Note that, to arrive at subproblem  $P(i)$ , we need to consider what happens if we take out an arbitrary coin. In particular, each coin will only add one to our tally. Therefore, to obtain the minimal number of coins, each previous subproblem needs to also maintain minimality. Therefore, we minimise each of the previous subproblems by considering all of the coin denominations; this gives us the following recursion

$$\text{OPT}(i) = \min \{\text{OPT}(i - v(k)) : 1 \leq k \leq n\} + 1,$$

for  $1 \leq i \leq C$ . To obtain the actual change sequence, suppose that the optimal  $k$  that was chosen was  $m$ . In other words,  $\text{OPT}(i) = \text{OPT}(i - v(m)) + 1$ . We can construct a table where index  $i$  stores the value  $m$ . Then backtracking to obtain the sequence is a matter of figuring out the right lookup based on the  $k$ th denomination for each subproblem.

We can define the base case to be  $\text{OPT}(0) = 0$  since no coins are required to obtain a value of 0. The final solution is  $\text{OPT}(C)$  and the natural order of computing the subproblems is in increasing order of  $i$ .

We finally justify the time complexity. There are  $C$  subproblems (one for each  $i$ ) and each subproblem requires a single loop through all of the  $n$  denominations. Therefore, each subproblem is solved in  $O(n)$  time, which implies that our algorithm runs in  $C \cdot O(n) = O(nC)$  time. It is important to note that  $C$  is not a constant, but rather part of our input. Therefore, we can't reduce this to  $O(n)$ .  $\square$

**[K] Exercise 6.** (*Edit Distance*) You are given string  $A$  of size  $n$  and string  $B$  of size  $m$ , and you want to transform  $A$  to  $B$ . You are allowed to insert a character, delete a character, and replace a character with another. However, each operation comes with a cost.

- Inserting a character costs  $c_I$ ;
- Deleting a character costs  $c_D$ ;
- Replacing a character costs  $c_R$ .

Design an  $O(mn)$  algorithm to find the lowest total cost to transform  $A$  into  $B$ .

How should you relate strings  $A$  and  $B$  together? Provide a suitable subproblem that allows you to do so.

*Solution.* The trick behind this exercise is to provide a natural subproblem that allows you to connect strings  $A$  and  $B$  together. In particular, we want to ensure that the first  $k$  letters of both strings are matching, so that we only need to change subsequent characters.

Therefore, we shall let  $P(i, j)$  be the subproblem: *what is the lowest total cost to transform the sequence  $A[1 \dots i]$  to  $B[1 \dots j]$ ?*. Let  $\text{OPT}(i, j)$  be the solution to subproblem  $P(i, j)$ .

To obtain the recursion, we need to find all of the ways to obtain  $A[1 \dots i]$  and  $B[1 \dots j]$ . If the current operation is delete, then we must have transformed  $A[1 \dots i - 1]$  into  $B[1 \dots j]$  and then deleted  $A[i]$ . If the current operation is insert, then we must have transformed  $A[1 \dots i]$  into  $B[1 \dots j - 1]$  and then appended  $B[j]$ . Otherwise, we must have transformed  $A[1 \dots i - 1]$  into  $B[1 \dots j - 1]$ . In this case, if  $A[i] = B[j]$ , there is nothing to do. Otherwise, if  $A[i] \neq B[j]$ , then we do a simple replacement. This defines our recursion as follows:

$$\text{OPT}(i, j) = \min \begin{cases} \text{OPT}(i - 1, j) + c_D, \\ \text{OPT}(i, j - 1) + c_I, \\ \text{OPT}(i - 1, j - 1) & \text{if } A[i] = B[j], \\ \text{OPT}(i - 1, j - 1) + c_R & \text{if } A[i] \neq B[j], \end{cases}$$

with  $1 \leq i \leq n$  and  $1 \leq j \leq m$ .

The base case is  $\text{OPT}(i, 0) = i \cdot c_D$  and  $\text{OPT}(0, j) = j \cdot c_I$ . The overall solution is  $\text{OPT}(n, m)$ . Note that there are  $mn$  subproblems (one for each character of  $A$  and one for each character of  $B$ ). However, updating each of the subproblem takes constant time. Therefore, the overall running time of the algorithm is  $O(mn)$ .  $\square$

**[K] Exercise 7.** A *palindrome* is a word that can be read the same both forwards and backwards. For example, the word “kayak” is a palindrome as reading it forwards is the same as reading it backwards. Similarly, “kaayak” is not a palindrome because reading it backwards reads “kayaak” which is not the same as “kaayak”. Given a string of length  $n$ , design an  $O(n^2)$  algorithm that finds the minimum number of characters to delete such that the resulting string after deletion is a palindrome.

For example, “kaayak” only requires one deletion to form a palindrome, while the string “abccab” requires two deletions.

*Solution.* Let  $s[1, \dots, n]$  be our input string.

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the minimum number of characters to delete such that the string  $s[i, i + 1, \dots, j]$  is a palindrome.
- **Recurrence:** Now, to derive a recursion, we first observe that  $\text{OPT}(i + 1, j - 1)$  gives us the smallest number of deletions required such that  $s[i + 1, \dots, j - 1]$  is a palindrome.

$$\underbrace{s[i + 1] s[i + 2] \dots s[j - 1]}_{\text{OPT}(i+1,j-1) \implies \text{palindrome}}$$

To ensure that  $s[i, \dots, j]$  is a palindrome, we now determine whether  $s[i] = s[j]$ . If so, then there are no extra deletions required to form a palindrome from index  $i$  to  $j$ . In other words, the number of deletions required is just the number of deletions that forms the palindrome  $s[i + 1, \dots, j - 1]$ ; in other words, we have that  $\text{OPT}(i, j) = \text{OPT}(i + 1, j - 1)$  in the case where  $s[i] = s[j]$ .

We now jump to the case where  $s[i] \neq s[j]$ . In this case, we need to delete one of  $s[i]$  or  $s[j]$ . To determine which character to delete, we compute both subproblems and take the minimum number of deletions among the two. Therefore, under the case where  $s[i] \neq s[j]$ , we have

$$\text{OPT}(i, j) = \min\{\text{OPT}(i + 1, j), \text{OPT}(i, j - 1)\} + 1.$$

In other words, the recurrence becomes

$$\text{OPT}(i, j) = \begin{cases} \text{OPT}(i + 1, j - 1) & \text{if } s[i] = s[j], \\ \min\{\text{OPT}(i + 1, j), \text{OPT}(i, j - 1)\} + 1 & \text{if } s[i] \neq s[j]. \end{cases}$$

For the edge case where  $j - 1 \leq i$ , set  $\text{OPT}(i, j) = 0$  since we already have a palindrome.

- **Base case:** For the base case, note that  $\text{OPT}(i, i) = 0$  for each  $1 \leq i \leq n$  since a single character is a palindrome.
- **Order of Computation and Final Solution:** The final solution is  $\text{OPT}(1, n)$ . To obtain the order of evaluation, observe that each subproblem relies on computing “smaller” length strings. Therefore, we should be solving the subproblems in increasing order of string size (i.e. in increasing order of  $j - i$ ).
- **Time Complexity:** To determine the time complexity, observe that there are  $n^2$  many subproblems. Each subproblem takes  $O(1)$  to compute; therefore, the running time of the algorithm is  $O(n^2)$ .

□

**[K] Exercise 8.** You are given a set of  $n$  types of rectangular boxes, where the  $i^{th}$  box has height  $h_i$ , width  $w_i$  and depth  $d_i$ . You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box. Design an  $O(n^2)$  algorithm that returns the maximum height of the stack of boxes.

How can we reduce this to a problem without rotations?

*Solution.* To simplify the problem, we distinguish among all of the rotations. For each box, there are six rotations since for each face, we can exchange its width and height and there are three faces. Thus, consider the problem of having  $6n$  types of rectangular boxes, which allows us to assume that there are no rotations involved. We now solve the original problem.

Using merge sort, we can order the boxes in decreasing order based on the surface area of their base (remember that each box is fixed and there are no rotations). In this way, if  $B_1$  can be stacked on top of  $B_0$ , then  $B_0$  must appear before  $B_1$  when we ordered the boxes.

- **Subproblem:** Let  $\text{OPT}(i)$  denote the maximum height possible for a stack if the top box is box number  $i$ .
- **Recurrence:** Observe that we look at *all* boxes whose base is strictly larger than box  $B_i$  and look at what the maximum height can be produced from placing box  $B_i$  at the top. That is, for each  $1 \leq i \leq 6n$ :

$$\text{OPT}(i) = \max\{\text{OPT}(j) + h_i : \text{over all } j \text{ such that } w_j > w_i, d_j > d_i\}.$$

- **Base case:**  $\text{OPT}(i) = h_i$ , for each  $i$  if we cannot place a box below  $B_i$ .
- **Order of Computation and Final Solution:** By focusing on the recurrence, we observe that each subproblem relies on smaller widths and smaller depth. Therefore, we compute the subproblems in increasing order of  $w_i$  and  $d_i$  (i.e. in increasing order of  $w_i + d_i$ ). The final solution is  $\max_{1 \leq i \leq 6n} \text{OPT}(i)$ .
- **Time Complexity:** Ordering the boxes using merge sort takes  $O(n \log n)$  time. For the dynamic programming component, we have  $6n$  subproblems and for each subproblem, we perform a  $O(n)$  search to find boxes whose base is large enough to stack the current box. Thus, we have an  $O(n \log n) + O(6n^2) = O(n^2)$  algorithm.

□

**[K] Exercise 9.** Given a sequence of  $n$  positive or negative integers  $A_1, A_2, \dots, A_n$ , determine a contiguous subsequence  $A_i$  to  $A_j$  for which the sum of elements in the subsequence is maximised.

*Solution.*

- **Subproblem:** Let  $\text{OPT}(i)$  denote the maximum sum of elements ending with integer  $i$ .

- **Recurrence:** Suppose that we have chosen the sequence of elements that end with  $i - 1$ . If we now consider the  $i$ th element of  $A$ , we can either choose to include it into our subsequence or  $A_i$  starts the new block.

  - If  $A_i + \text{OPT}(i - 1)$  is negative, then  $A_i$  should start a new chain.
  - If  $A_i + \text{OPT}(i - 1)$  is non-negative, then it is safe to keep  $A_i$  as part of the current chain.

Therefore, we obtain the recurrence

$$\text{OPT}(i) = A_i + \max\{\text{OPT}(i - 1), 0\}.$$

We can determine the contiguous subsequence by keeping track of the starting element of the chain. To  $\text{OPT}(i - 1) \leq 0$ , then the  $i$ th element forms the start of the chain. Therefore, define  $\text{start}(i)$  to denote the start of the chain that includes the  $A_i$ . Then, for all  $1 \leq i \leq n$ , we have the recurrence

$$\text{start}(i) = \begin{cases} \text{start}(i - 1) & \text{if } \text{OPT}(i - 1) > 0, \\ i & \text{if } \text{OPT}(i - 1) \leq 0. \end{cases}$$

- **Base case:** If there are no elements chosen, then the maximum sum of the elements is 0. Therefore,  $\text{OPT}(0) = 0$ .
- **Order of Computation and Final Solution:** We observe that the recurrence lies on smaller values of  $i$ . Therefore, the natural order of evaluation is in increasing order of  $i$  with the final solution being  $\max_{1 \leq i \leq n} \text{OPT}(i)$ . We can return the contiguous subproblem by returning the interval  $[\text{start}(i), i]$  for the given index  $i$ .
- **Time Complexity:** There are  $n$  subproblems and each subproblem takes  $O(1)$  to compute; therefore, the running time of the algorithm is  $O(n)$ .

□

**[K] Exercise 10.** Due to the recent droughts,  $n$  proposals have been made to dam the Murray river. The  $i$ th proposal asks to place a dam  $x_i$  meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within  $r_i$  metres (upstream or downstream). Design an algorithm that returns the maximal number of dams that can be built, you may assume that  $x_i < x_{i+1}$  for all  $i = 1, \dots, n - 1$ .

*Solution.*

- **Subproblem:** Let  $\text{OPT}(i)$  denote the maximum number of dams that can be built among proposals  $1, \dots, i$  such that the  $i$ th dam is built.
- **Recurrence:** We first observe that, if we build the  $i$ th dam, then we cannot place any dam where  $|x_i - x_j| \leq r_i$  and  $|x_i - x_j| \leq r_j$ . Therefore, we need to look at all dams such that  $|x_i - x_j| > \max\{r_i, r_j\}$ . By our subproblem formulation, we look at all proposals from  $1, \dots, i$ . Therefore, we look at all possible dams  $j < i$  that satisfy the above constraint that maximises the number of dams that we can build. This is equivalent to building dam  $i$  and then building dams using proposals  $1, \dots, j$ , giving us the recurrence

$$\text{OPT}(i) = 1 + \max_{j < i} \{\text{OPT}(j) : x_i - x_j > \max(r_i, r_j)\}.$$

- **Base case:** If there is only one dam, then we build the dam. Therefore,  $\text{OPT}(1) = 1$ .
- **Order of Computation:** To obtain the appropriate recurrence, we need to look at all proposals from  $1, \dots, j, \dots, i$ . Therefore, the natural order of evaluation is in increasing order of  $i$ .
- **Overall answer:** Considering all possible choices for the last dam to be built, the answer is

$$\max_{1 \leq i \leq n} \text{opt}(i).$$

- **Time Complexity:** There are  $n$  subproblems. However, each subproblem requires us to look at  $j < i$  subproblems. This gives  $1 + 2 + \dots + n = O(n^2)$  many subproblems to compute altogether. Therefore, the running time of the algorithm is  $O(n^2)$ .

□

**[K] Exercise 11.** You are travelling by canoe down a river and there are  $n$  trading posts along the way. Before starting your journey, you are given for each  $1 \leq i < j \leq n$  the fee  $F(i, j)$  for renting a canoe from post  $i$  to post  $j$ . These fees are arbitrary, for example it is possible that  $F(1, 3) = 10$  and  $F(1, 4) = 5$ . You begin at trading post 1 and must end at trading post  $n$  (using rented canoes). Your goal is to design an efficient algorithm that produces the sequence of trading posts where you change your canoe which minimizes the total rental cost.

*Solution.*

- **Subproblem:** Let  $\text{OPT}(i)$  denote the minimum cost it would take to reach post  $i$ .
- **Recurrence:** We note that, to arrive at post  $i$ , we must come from some post  $j$ . We look at all possible posts that we can come from and choose the post that minimises the overall cost. This gives us

$$\text{OPT}(i) = \min_{1 \leq j \leq i} \{\text{OPT}(j) + F(j, i)\}.$$

- **Base case:** We begin at trading post 1; therefore, the base case is  $\text{OPT}(1) = 0$ .
- **Order of Computation and Final Solution:** Since our subproblem recurrence relies on previous values of  $i$ , the natural ordering is in increasing order of  $i$  with the final solution being  $\text{OPT}(n)$ .

We then reconstruct the sequence of trading posts the canoe had to have visited. For  $i > 1$  we define the following function:

$$\text{from}(i) = \underset{1 \leq j < i}{\operatorname{argmin}} \{\text{OPT}(j) + F(j, i)\},$$

where  $\operatorname{argmin}$  returns the value  $j$  that minimises  $\text{OPT}(j) + F(j, i)$ . To obtain the sequence, we backtrack from  $n$ , giving the sequence:

$$\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$$

and reversing this sequence gives the boat's journey.

- **Time Complexity:** There are  $n$  subproblems and each subproblem takes  $O(n)$  time to find the previous trading post. Therefore, the overall running time is  $O(n^2)$ .

□

**[K] Exercise 12.** You have an amount of money  $M$  and you are in a candy store. There are  $n$  kinds of candies and for each candy, you know how much pleasure you get by eating it, which is a number between 1 and  $K$ , as well as the price of each candy. Your task is to choose which candies you are going to buy to maximise the total pleasure you will get by gobbling them all.

*Solution.* This is a knapsack problem with duplicated values. The pleasure score is the value of the item, the cost of a particular type of candy is its weight, and the money  $M$  is the capacity of the knapsack. The complexity is  $O(Mn)$ .

□

**[K] Exercise 13.** Your shipping company has just received  $N$  shipping requests (jobs). For each request  $i$ , you know it will require  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $T$  trucks in total. Out of these  $N$  jobs you can take as many as you would like, as long as no more than  $T$  trucks are used total. Devise an efficient algorithm to select jobs that will bring you the largest possible amount of money.

*Solution.* We can recognise that this is nothing but the standard knapsack problem with  $t_i$  being the size of the  $i$ th item,  $d_i$  its value and with  $T$  as the capacity of the knapsack. Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed. The complexity follows the knapsack problem which results to be  $O(NT)$ .  $\square$

**[K] Exercise 14.** Again your shipping company has just received  $N$  shipping requests (jobs). This time, for each request  $i$ , you know it will require  $e_i$  employees and  $t_i$  trucks to complete, paying you  $d_i$  dollars. You have  $E$  employees and  $T$  trucks in total. Out of these  $N$  jobs you can take as many of them as you would like, as long as no more than  $E$  employees and  $T$  trucks are used in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

*Solution.* This is a slight modification of the knapsack problem with two constraints on the total size of all jobs; think of a knapsack which can hold items of total weight not exceeding  $E$  units of weight and total volume not exceeding  $T$  units of volume, with item  $i$  having a weight of  $e_i$  integer units of weight and  $t_i$  integer units of volume.

- **Subproblem:** For each triplet  $e \leq E, t \leq T, i \leq N$ , let  $\text{OPT}(i, e, t)$  denote the subcollection of items  $1, \dots, i$  that fits in a knapsack of capacity  $e$  and  $t$  units of volume which is of largest possible value.
- **Recurrence:** The recurrence is then as follows:

$$\text{OPT}(i, e, t) = \max\{\text{OPT}(i - 1, e, t), \text{OPT}(i - 1, e - e_i, t - t_i) + d_i\},$$

and the rest follows as with the knapsack problem. The time complexity is  $O(NET)$ .

$\square$

**[H] Exercise 15.** There are  $m$  levels in a video game, and in each level, there are  $n$  doors you can choose from. Each door has a treasure chest with value  $a_{i,j}$ , where  $a_{i,j}$  represents the treasure value of door  $j$  in level  $i$ . In each level, you can only choose one door and once you choose that particular door, you can only choose from doors directly adjacent to the chosen door (if they exist) in the next level. In other words, once you choose door  $i$  in level  $j$ , then you can only choose between doors  $i - 1, i$  or  $i + 1$  on level  $j + 1$ . On level 1, you can choose any door.

Design an  $O(mn)$  algorithm to maximise the total value accrued by choosing one door in each of the  $m$  levels.

*Solution.*

- **Subproblem:** Let  $\text{possible}(i, k)$  denote whether the player can reach level  $i$  after playing exactly  $k$  levels.
- **Recurrence:** To reach level  $i$  using exactly  $k$  levels, we can either arrive to level  $i$  directly from level  $i - 1$  using  $k - 1$  levels previously, or arrive from a door at some other level  $j < i - 1$  using exactly  $k - 1$  levels. To make our recurrence easier to compute, we define the following predicate  $\text{link}(j, i)$  to denote whether level  $j$  has a secret exit to level  $i$ . That is,  $\text{link}(j, i)$  is true if and only if level  $j$  has a secret exit to level  $i$ . The recurrence then becomes

$$\text{possible}(i, k) = \text{possible}(i - 1, k - 1) \vee \left( \bigvee_{1 \leq j < i - 1} \text{possible}(j, k - 1) \wedge \text{link}(j, i) \right).$$

Note that the right hand side of the recurrence only suggests that there must be *some* level that can be reached using only  $k - 1$  levels and has a secret exit to level  $i$ .

- **Base case:** Since reaching 0 levels is trivially true, we have that  $\text{possible}(0, 0) = \text{true}$ .
- **Order of Computation and Final Solution:** Since we need to compute all levels for each value of  $k$ , we solve each subproblem in increasing order of  $i$  and then increasing order of  $k$ , with the final solution being  $\text{possible}(K, n)$ .
- **Time Complexity:** There are  $O(nK)$  subproblems in total, and it appears at first that we need to iterate through all  $j < i$  for each one. However, we can do better. Since only  $n$  links exist, we can precompute for each level  $i$  which earlier levels  $j$  link to it, and iterate through only this list. In this way, all subproblems  $O(\cdot, k)$  take a total of  $O(n)$  time, and therefore the total time complexity is  $O(nK)$ .

□

**[H] Exercise 16.** You have  $n_1$  items of size  $s_1$  and  $n_2$  items of size  $s_2$ . You would like to pack all of these items into bins, each of capacity  $C > s_1, s_2$ , using as few bins as possible. Design an algorithm that returns the minimal number of bins required.

*Solution.*

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the minimum number of bins required to pack  $i$  many items of size  $s_1$  and  $j$  many items of size  $s_2$ .
- **Recurrence:** Let  $K$  denote the ratio  $C/s_1$ . This denotes the maximum number of items we can pack into one bin using items of size  $s_1$ . We find the value  $k$  such that  $k$  items of size  $s_1$  can be placed in one bin. We also find the value  $k$  such that we can fill in as many items of size  $s_2$  into the same bin. This gives

$$\text{OPT}(i, j) = 1 + \min_{1 \leq k \leq K} \left\{ \text{OPT} \left( i - k, j - \left\lfloor \frac{C - ks_1}{s_2} \right\rfloor \right) \right\}.$$

- **Base case:** We can compute the base case of  $\text{OPT}(1, 1)$  by considering the values of  $s_1, s_2, C$ .
- **Order of Computation and Final Solution:** We compute each of the subproblems in increasing order of  $j$  and  $i$ .
- **Time Complexity:** There are  $K$  many placements for each  $(i, j)$  pair. Therefore, the running time is  $O(K \cdot n_1 n_2) = O(C \cdot n_1 n_2)$ .

□

**[H] Exercise 17.** We are given a checkerboard which has 4 rows and  $n$  columns, and has an integer written in each square. We are also given a set of  $2n$  pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximise the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

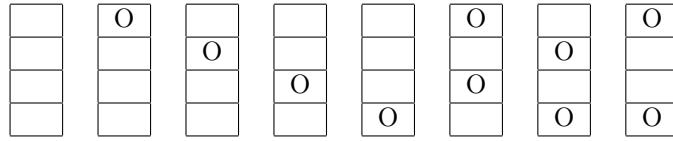
- (a) Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider sub-problems consisting of the first  $k$  columns  $1 \leq k \leq n$ . Each sub-problem can be assigned a type, which is the pattern occurring in the last column.

- (b) Using the notions of compatibility and type, give an  $O(n)$ -time algorithm for computing an optimal placement.

*Solution.*

- (a) There are 8 patterns, listed below.



- (b) Let  $t$  denote the type of pattern so that  $1 \leq t \leq 8$ .

- **Subproblem:** Let  $\text{OPT}(k, t)$  denote the maximum score that can be achieved by only placing pebbles in the first  $k$  columns such that the  $k$ th column contains pattern  $t$ .
- **Recurrence:** We first define  $\text{score}(k, t)$  to be the score obtained by using pattern  $t$  on column  $k$ . Then the recurrence becomes

$$\text{OPT}(k, t) = \text{score}(k, t) + \max \{ \text{OPT}(k - 1, s) : s \text{ is compatible with } t \}.$$

- **Base case:** If there is no column, then any pattern on the column gives a score of 0. Therefore,  $\text{OPT}(0, t) = 0$  for each  $1 \leq t \leq 8$ .
- **Order of Computation and Final Solution:** For each column, we need to solve for every pattern. Therefore, we solve in increasing order of  $t$  and then increasing order of  $k$ , with the final solution being  $\max_{1 \leq t \leq 8} \text{OPT}(n, t)$ .
- **Time Complexity:** Since the number of patterns is fixed, there are only  $O(n)$  many subproblems and each subproblem only needs at most eight many subproblems to check. Therefore, the running time is  $O(n)$ .

□

**[H] Exercise 18.** A company is organising a party for its employees. The organisers of the party want it to be a fun party, and so have assigned a fun rating to every employee. The employees are organised into a strict hierarchy, i.e. a tree rooted at the president. There is one restriction, though, on the guest list to the party: an employee and their immediate supervisor (parent in the tree) cannot both attend the party (because that would be no fun at all). Give an algorithm that makes a guest list for the party that maximises the sum of the fun ratings of the guests.

*Solution.* Suppose that  $T(i)$  defines the subtree of the tree  $T$  of all employees that is rooted by employee  $i$ . Then, for each subtree, we will look at the maximum sum of the fun ratings in two ways: one which includes employee  $i$  and one that does not include employee  $i$ .

For each subtree, we define the following quantities:

- $I(i)$  is the maximal sum of fun factors  $\text{fun}(i)$  that satisfies all of the constraints but *includes* root  $i$ ,
- $E(i)$  is the maximal sum of fun factors  $\text{fun}(i)$  that satisfies all of the constraints but *excludes* root  $i$ .

With these quantities defined, we now compute the dynamic programming solution.

- **Subproblem:** Let  $\text{OPT}(i)$  denote the maximum sum of the fun ratings of  $T(i)$ .
- **Recurrence:** For the recurrence, we need to compute two quantities,  $I(i)$  and  $E(i)$ . For each non-leaf node, we need to consider excluding the subordinates (because  $i$  is the immediate supervisor of these workers) if we choose to include employee  $i$ , or if we exclude  $i$ , then we can either choose to exclude the children or include the children of  $i$ , whichever value is greater. Thus, we compute  $I(i)$  by

$$I(i) = \text{fun}(i) + \sum_{1 \leq k \leq m} E(j_k),$$

where  $j_1, \dots, j_m$  are the subordinates of  $i$ .

We compute  $E(i)$  by

$$E(i) = \sum_{1 \leq k \leq m} \max(I(j_k), E(j_k)),$$

where  $j_1, \dots, j_m$  are defined as above. In this way, *for each* child of  $i$ , we can either include them or exclude them.

Then, we have that

$$\text{OPT}(i) = \max(I(i), E(i)).$$

- **Base case:** For each  $i$  that is a leaf node,  $\text{OPT}(i) = \text{fun}(i)$ .
- **Order of Computation and Final Solution:** We solve the subproblems from the leaves of the tree to the root of the leaves, where the final solution is simply  $\text{OPT}(n) = \max(I(n), E(n))$  where  $n$  is the root of the tree.
- **Time Complexity:** The time complexity is linear in the number of employees because we only need to scan through each of the employees once. Each employee only has one parent. In the worst case, we need to ask every employee exactly once.

□

**[H] Exercise 19.** You have to cut a wood stick into several pieces. The most affordable company, Analog Cutting Machinery (ACM), charges money according to the length of the stick being cut. Their cutting saw allows them to make only one cut at a time. It is easy to see that different cutting orders can lead to different prices. For example, consider a stick of length 10 m that has to be cut at 2, 4, and 7 m from one end. There are several choices. One can cut first at 2, then at 4, then at 7. This leads to a price of  $10 + 8 + 6 = 24$  because the first stick was of 10 m, the resulting stick of 8 m, and the last one of 6 m. Another choice could cut at 4, then at 2, then at 7. This would lead to a price of  $10 + 4 + 6 = 20$ , which is better for us. Your boss demands that you design an  $O(n^2)$  algorithm to find the minimum possible cutting cost for any given stick.

*Solution.* Define  $x(i)$  to be the distance along the stick where the  $i$ th cut occurs. We, then, approach with dynamic programming.

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the minimum cost of cutting up the stick from the cutting point  $i$  to cutting point  $j$ .
- **Recurrence:** Choosing to cut at points  $i$  and  $j$  will decrease the distance to  $x(j) - x(i)$ . In this interval, we then need to find the smallest possible cut(s) in terms of cost; therefore, we need to compute  $\text{OPT}(i, k) + \text{OPT}(k, j)$  for each  $i < k < j$ . Thus, we have

$$\text{OPT}(i, j) = x(j) - x(i) + \min \{ \text{OPT}(i, k) + \text{OPT}(k, j) : 1 < k < j \}.$$

- **Base case:** Fixing  $i$ , the base case occurs when  $j = i + 1$  in which case,  $\text{OPT}(i, i + 1) = 0$  for each  $i$ .
- **Order of Computation and Final Solution:** We solve each subproblem in increasing order of  $j$  and then increasing order of  $i$  with the final solution being  $\text{OPT}(1, n)$ .
- **Time Complexity:** For each  $i$ , we need to solve  $n - 1$  subproblems. Thus, there are  $O(n^2)$  many subproblems, each of which take constant time. Thus, the time complexity is  $O(n^2)$ .

□

**[H] Exercise 20.** You have been handed responsibility for a business in Texas for the next  $n$  days. Initially, you have  $K$  illegal workers. At the beginning of each day, you may hire an illegal worker, keep the number of illegal workers the same or fire an illegal worker. At the end of each day, there will be an inspection. The inspector on the  $i^{th}$  day will check that you have between  $l_i$  and  $r_i$  illegal workers (inclusive). If you do not, you will fail the inspection. Design an  $O(n^2)$  algorithm that determines the fewest number of inspections you will fail if you hire and fire illegal employees optimally.

*Solution.* Since we can either hire or fire an illegal worker (or do nothing), then we observe that the maximum on the evening of day  $i$  can be obtained by continuously hiring an illegal worker whereas the minimum on day  $i$  can be obtained by continuously firing an illegal worker. Therefore, the maximum number of illegal workers is given by  $K + i$  while the minimum number of illegal workers is given by  $\max(K - i, 0)$ . We will use this as part of our recurrence.

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the minimum number of inspections to fail such that there are  $j$  many illegal workers on day  $i$ .
- **Recurrence:** We first require that  $\max(0, K - i) \leq j \leq K + i$ . To help us with the recurrence, denote  $\text{failed}(i, j)$  to indicate whether  $l_i \leq j \leq r_i$ . That is,

$$\text{failed}(i, j) = \begin{cases} 0 & \text{if } l_i \leq j \leq r_i, \\ 1 & \text{otherwise.} \end{cases}$$

Then we either need to consider hiring an illegal worker, firing an illegal worker or do nothing. We look at each of the cases separately to give the recurrence

$$\text{OPT}(i, j) = \text{failed}(i, j) + \min \begin{cases} \text{OPT}(i - 1, j - 1) & \text{if } j - 1 \leq \max(0, K - (i - 1)), \\ \text{OPT}(i - 1, j), \\ \text{OPT}(i - 1, j + 1) & \text{if } j + 1 \leq K + (i - 1). \end{cases}$$

- **Base case:** On the zeroth day, we have no failed inspections. Therefore, we obtain  $\text{OPT}(0, K) = 0$ .
- **Order of Computation and Final Solution:** Observe that our recurrence relies on all possible values of  $j$ , for each day  $i$ . Therefore, we solve our subproblems in increasing order of  $j$  and then in increasing order of  $i$ , with the final solution being

$$\min\{\text{OPT}(n, j) : \max(K - n, 0) \leq j \leq K + n\}.$$

- **Time Complexity:** There are at most  $n$  days and  $2n + 1$  possibility of illegal workers since there needs to be a subproblem for all workers between  $K - n \leq j \leq K + n$ . Therefore, there are  $n(2n + 1) = O(n^2)$  subproblems, each of which is computed in  $O(1)$  time. Thus, the overall running time of the algorithm is  $O(n^2)$ .

□

## § SECTION THREE: COUNTING

[K] **Exercise 21.** Given an array of  $n$  positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most  $k$  in  $O(n^2)$  time.

*Solution.*

- **Subproblem:** Let  $\text{NUM}(i)$  denote the number of ways to split the first  $i$  elements into contiguous blocks of sum at most  $k$ .
- **Recurrence:** We look at all of the possible ways to place a divider such that all elements in the same group of the divider sum to at most  $k$ . Denote  $\text{sum}(j, i)$  to be the sum of all elements between (and including)  $j$  and  $i$ . Then we have that

$$\text{NUM}(i) = \sum_{\substack{1 \leq j \leq i, \\ \text{sum}(j, i) \leq k}} \text{NUM}(j - 1).$$

- **Base case:** There is exactly one way to split 0 elements; therefore,  $\text{NUM}(0) = 1$ .
- **Order of Computation and Final Solution:** We solve the subproblems in increasing order of  $i$ . For a particular subproblem  $\text{NUM}(i)$ , we consider the  $j$  values in *decreasing* order, so that we can update  $\text{sum}(j, i)$  value in constant time by adding one more term at each step until the early exit. The final solution is  $\text{NUM}(n)$ .
- **Time Complexity:** There are  $n$  subproblems and each subproblem requires an  $O(n)$  search. Therefore, the running time of the algorithm is  $O(n^2)$ .

□

[K] **Exercise 22.** Let  $A$  and  $B$  be two strings of lengths  $n$  and  $m$  respectively. We say that a string  $A$  occurs as a subsequence of another string  $B$  if we can obtain  $A$  by deleting some of the letters of  $B$ . Given strings  $A$  and  $B$ , design an  $O(mn)$  algorithm that gives the number of different occurrences of  $A$  in  $B$ ; that is, we want to compute the number of ways one can delete some of the symbols of  $B$  to get  $A$ .

*Solution.* We begin by defining some notations. Let  $A_i$  denote the  $i$ th letter of  $A$  and  $B_j$  as the  $j$ th letter of  $B$ .

- **Subproblem:** Let  $\text{ways}(i, j)$  denote the number of times the first  $i$  letters of  $A$  appears as a subsequence in the first  $j$  letters of  $B$ .
- **Recurrence:** If  $A_i \neq B_j$ , then this is equivalent to removing the  $j$ th character from  $B$  and checking if the first  $i$  characters appear in the substring  $B_1, \dots, B_{j-1}$ . This gives  $\text{ways}(i, j - 1)$  number of ways this could happen.

On the other hand, if  $A_i = B_j$ , then we check to see if  $A_{i-1} = B_{j-1}$ . This gives  $\text{ways}(i - 1, j - 1)$  number of ways. However, the substring  $A$  ending at  $A_i$  can also be contained entirely inside the string  $B_1, \dots, B_{j-1}$ . Therefore, we also compute  $\text{ways}(i, j - 1)$ . In other words, we obtain the recurrence

$$\text{ways}(i, j) = \begin{cases} \text{ways}(i, j - 1) & \text{if } A_i \neq B_j, \\ \text{ways}(i - 1, j - 1) + \text{ways}(i, j - 1) & \text{if } A_i = B_j. \end{cases}$$

- **Base case:** If  $A$  is empty, then  $\text{ways}(0, j) = 1$  for each  $1 \leq j \leq m$ . If  $B$  is empty, then  $\text{ways}(i, 0) = 0$  for each  $1 \leq i \leq n$ .
- **Order of Computation and Final Solution:** We solve each subproblem in increasing order of  $j$  and then increasing order of  $i$ .

- **Time Complexity:** There are  $mn$  many subproblems (one for  $i$  and one for  $j$ ) and each subproblem takes constant time to compute; therefore, the running time of the algorithm is  $O(mn)$ .

□

**[K] Exercise 23.** A partition of a number  $n$  is a sequence  $\langle p_1, p_2, \dots, p_k \rangle$  such that  $1 \leq p_1 \leq p_2 \leq \dots \leq p_k \leq n$ . We call each  $p_i$  a *part*. Define  $\text{NUMPART}(n, k)$  to be the number of partitions of  $n$  such that each part is at most  $k$ .

- Explain why  $\text{NUMPART}(n, 1) = 1$  for each  $n$ .
- Devise a recurrence to determine the number of partitions of  $n$  in which every part is at most  $k$ , where  $n, k$  are two given integers such that  $2 \leq k \leq n$ .
  - To obtain the right recursion, consider two different cases depending on the largest possible value of each part.
- Hence, find the total number of partitions of  $n$ .
- How many subproblems are there, and for each subproblem, what is the time complexity to compute? Hence, analyse the time complexity of the algorithm.

*Solution.*

- Note that  $\text{NUMPART}(n, 1)$  counts the number of ways of expressing  $n$  such that every part is at most 1. This can be done precisely when every part is 1.
- We split up the partitions by the value of their largest part  $p_t$ . More precisely, we consider the cases where  $p_t = k$  and  $p_t < k$  separately.

For the first case, we have the equation

$$p_1 + \dots + p_{t-1} + k = n,$$

and subtracting  $k$  from both sides gives

$$p_1 + \dots + p_{t-1} = n - k.$$

We observe that the remaining sequence  $\langle p_1, \dots, p_{t-1} \rangle$  is a partition of  $n - k$  in which no part exceeds  $k$ , so there are  $\text{NUMPART}(k, n - k)$  such sequences.

For the second case, if  $p_t < k$  then every other term of the sequence is also less than  $k$ . Therefore  $\langle p_1, \dots, p_t \rangle$  is a partition of  $n$  in which no part exceeds  $k - 1$ , so there are  $\text{NUMPART}(n, k - 1)$  such sequences.

Note that these cases will not overlap (i.e. you can't have a case where the greatest part is  $k$  and not  $k$  simultaneously). Therefore, the number of partitions is simply

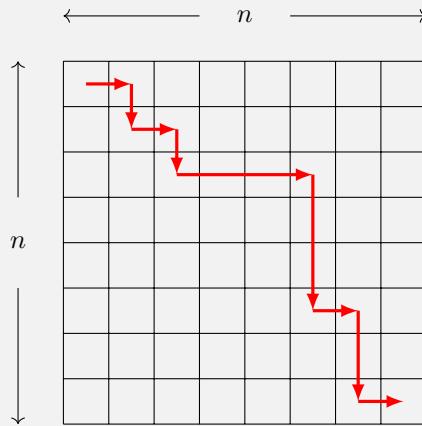
$$\text{NUMPART}(n, k) = \text{NUMPART}(n - k, k) + \text{NUMPART}(n, k - 1).$$

We can solve these subproblems in lexicographic order, i.e. increasing order of  $k$ , breaking ties in increasing order of  $n$ .

- If we allow  $p_t$  to take any value up to  $n$ , then any partition of  $n$  is allowed. Therefore the total number of (unrestricted) partitions of  $n$  is simply  $\text{NUMPART}(n, n)$ .
- Note that there is one subproblem for each number of partitions and one subproblem for the maximum size of each partition. Therefore, there are  $n^2$  many subproblems. For each subproblem, it takes  $O(1)$  time to compute. Thus, the overall running time is  $O(n^2)$ .

□

**[H] Exercise 24.** You are given an  $n \times n$  chessboard with an integer in each of its  $n^2$  squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.



- (a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.
- (b) Describe an algorithm which always correctly finds a minimal sum path and runs in time  $n^2$ .
- (c) Describe an algorithm which computes the number of such minimal paths.

*Solution.*

- (a) Consider the following greedy solution. Starting at the top left square, we move to the square below or to the right that contains the smallest integer.

Using this solution, consider the following counterexample.

|   |      |      |
|---|------|------|
| 0 | 1    | 1    |
| 5 | 1000 | 1000 |
| 5 | 5    | 0    |

The algorithm would choose the path 1-1-1000-0 for a score of 1002, while the correct path would be 5-5-5-0 for a score of 15.

- (b) We define the quantity  $\text{BOARD}(i, j)$  to be the integer inside the cell located on row  $i$  and column  $j$ . We then begin with the dynamic programming approach.

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the best score that can be achieved if we arrive at cell  $(i, j)$ .

- **Recurrence:** If we land at cell  $(i, j)$ , observe that the previous move must have been either  $(i - 1, j)$  (i.e. we move to the right to land at the cell) or  $(i, j - 1)$  (i.e. we move downwards to land at the cell). Thus, we need to check which way yielded us with the minimum value *so far*. Another way to see this is that, if we land at the final square  $(n, n)$ , then the path that we would need to trace needs to be the path where it is minimal ending at either cell  $(n - 1, n)$  or  $(n, n - 1)$ , and we repeat this process until we land back at the original square.

Thus, the recursion is

$$\text{OPT}(i, j) = \text{BOARD}(i, j) + \min \{\text{OPT}(i - 1, j), \text{OPT}(i, j - 1)\}.$$

- **Base case:** The base case is  $\text{OPT}(1, 1) = \text{BOARD}(1, 1)$ . To ensure that we never consider illegal moves (i.e. moves that land outside of the board), we also have that  $\text{OPT}(i, j) = \infty$  if cell  $(i, j)$  does not exist (or is off the board).
- **Order of Computation and Final Solution:** Observe that, in order to solve  $\text{OPT}(i, j)$ , we need to know  $\text{OPT}(i - 1, j)$  and  $\text{OPT}(i, j - 1)$ . Thus, the order in which we solve the problem is that we solve the subproblems in increasing order of  $i$ , and then increasing order of  $j$  to break ties. The final solution is simply solving  $\text{OPT}(n, n)$ . To return the actual path, we can make a note of the choice that the recursion makes by choosing either  $\text{OPT}(i - 1, j)$  or  $\text{OPT}(i, j - 1)$  by backtracking through the subproblems that are chosen. This gives us a path from  $(n, n)$  to  $(1, 1)$  which yields the path taken.
- **Time Complexity:** To analyse the time complexity, we require two pieces of information: the number of subproblems and the time taken to compute each subproblem. There are  $n^2$  subproblems for each  $1 \leq i, j \leq n$ . Now, for each subproblem  $P(i, j)$ , we would have computed  $\text{OPT}(i - 1, j)$  and  $\text{OPT}(i, j - 1)$  previously. Hence, we have  $O(1)$  lookup time and the computation for  $\text{OPT}(i, j)$  is simply a comparison and an arithmetic operation, all of which takes  $O(1)$  time. Thus, the overall time complexity is  $n^2 \cdot O(1) = O(n^2)$ .

(c) We approach this similar to part (b); in fact, we will use  $\text{OPT}(i, j)$  as defined above to our advantage.

- **Subproblem:** Let  $\text{WAYS}(i, j)$  denote the minimum number of ways to reach cell  $(i, j)$  with score  $\text{OPT}(i, j)$ .
- **Recurrence:** In much the same way, if we land at cell  $(i, j)$ , the previous move must have been either from  $(i - 1, j)$  or  $(i, j - 1)$ . We, therefore, need to check the minimum path from both of these directions. There are three cases to consider:
  - If the minimum path happens to come from  $(i - 1, j)$ , then we ignore all of the paths from  $(i, j - 1)$  because those paths will not be minimal.
  - If the minimum path happens to come from  $(i, j - 1)$ , then we ignore all of the paths from  $(i - 1, j)$ .
  - However, if they are equal, then we consider *all* possible paths to land at  $(i, j)$ .

With this in mind, it is easy to see that the recurrence becomes

$$\text{WAYS}(i, j) = \begin{cases} \text{WAYS}(i - 1, j) & \text{if } \text{OPT}(i - 1, j) < \text{OPT}(i, j - 1), \\ \text{WAYS}(i, j - 1) & \text{if } \text{OPT}(i - 1, j) > \text{OPT}(i, j - 1), \\ \text{WAYS}(i - 1, j) + \text{WAYS}(i, j - 1) & \text{if } \text{OPT}(i - 1, j) = \text{OPT}(i, j - 1). \end{cases}$$

- **Base case:** The base case is  $\text{WAYS}(1, 1) = 1$  since there is only one way to get to cell  $(1, 1)$ . We also define  $\text{WAYS}(i, j) = 0$  for all cells that fall outside of the grid since there are no ways to get to those cells.
- **Order of Computation and Final Solution:** We solve the problems in increasing order of  $i$ , and then increasing order of  $j$  to break ties for the same reason as the previous part. The final solution is  $\text{WAYS}(n, n)$ .
- **Time Complexity:** The time complexity is  $O(n^2)$  since there are still  $n^2$  subproblems and each subproblem takes  $O(1)$  time to compute.

□

## § SECTION FOUR: GAMES AND GRAPHS

[K] **Exercise 25.** Consider two deck of  $n$  and  $m$  cards respectively. A new deck of cards is formed out of the  $n + m$  cards. We say that the new deck is an *interleaving* if the ordering of the two original decks are preserved in the new deck. Given the two deck of cards and the new deck, say  $X, Y, Z$ , respectively, design an  $O(nm)$  algorithm that determines if  $Z$  is an interleaving of  $X$  and  $Y$ .

*Solution.*

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote whether the first  $i$  cards of  $X$  and the first  $j$  cards of  $Y$  form an interleaving of the first  $i + j$  cards of  $Z$ .
- **Recurrence:** First, denote  $X_i, Y_i, Z_i$  to be the  $i$ th card from decks  $X, Y$ , and  $Z$  respectively. We note that, if  $\text{OPT}(i, j)$  forms an interleaving, then we need to check which deck of cards come first.
  - If  $X$  comes first, then the  $i$ th card of  $X$  must form the  $(i + j)$ th card of  $Z$  and the first  $i - 1$  cards from  $X$  forms an interleaving with the first  $j$  cards from  $Y$ . This gives the recurrence

$$\text{OPT}(i, j) = \text{OPT}(i - 1, j) \wedge (Z_{i+j} = X_i).$$

- If  $Y$  comes first, then the  $j$ th card of  $Y$  must form the  $(i + j)$ th card of  $Z$  and the first  $j - 1$  cards from  $Y$  forms an interleaving with the first  $i$  cards from  $X$ . This gives the recurrence

$$\text{OPT}(i, j) = \text{OPT}(i, j - 1) \wedge (Z_{i+j} = Y_j).$$

This gives the recurrence

$$\text{OPT}(i, j) = (\text{OPT}(i - 1, j) \wedge (Z_{i+j} = X_i)) \vee (\text{OPT}(i, j - 1) \wedge (Z_{i+j} = Y_j)).$$

- **Base case:** Our base case occurs when we reach the empty string, giving us  $\text{OPT}(0, 0) = \text{true}$ . For any  $i < 0$  or  $j < 0$ , we also set  $\text{OPT}(i, j) = \text{false}$ .
- **Order of Computation and Final Solution:** We compute the subproblems in increasing order of  $j$  and then  $i$ , with the final solution being  $\text{OPT}(m, n)$ .
- **Time Complexity:** We have  $mn$  many subproblems and each takes constant time to compute; therefore, our algorithm takes  $O(mn)$  running time.

□

[K] **Exercise 26.** Some people think that the bigger an elephant is, the smarter it is. To disprove this, you want to analyse a collection of elephants and place as large a subset of elephants as possible into a sequence whose weights are increasing but their IQ's are decreasing. Design an  $O(n \log n)$  algorithm which, given the weights and IQ's of  $n$  elephants, will find a longest sequence of elephants such that their weights are increasing but IQ's are decreasing.

*Solution.* Using merge sort (or equivalent), sort the elephants in decreasing order of IQ. This problem now reduces to the longest increasing subsequence problem in terms of the weight of the elephants. Sorting takes  $O(n \log n)$  and the longest increasing subsequence problem also takes  $O(n \log n)$ . Hence, the overall time complexity is  $O(n \log n)$ . □

[K] **Exercise 27.** Let  $G = (V, E)$  be a directed and weighted graph on  $n$  vertices. In other words,  $|V| = n$ . For two vertices  $u, v \in V$ , we define the *distance*  $\delta(u, v)$  from  $u$  to  $v$  to be the weight of the shortest path from  $u$  to  $v$ . However, we alter the distances from  $u$  to  $v$  for two special cases:

- If there is no path from  $u$  to  $v$ , define  $\delta(u, v) = \infty$ .
- If there is a path from  $u$  to  $v$  that contains a negative-weight cycle, define  $\delta(u, v) = -\infty$ .

Given a graph  $G$ , design an  $O(n^3)$  algorithm that computes  $\delta(u, v)$  for all pairs  $u, v \in V$ .

Note that Floyd-Warshall's algorithm assumes that  $G$  contains no negative-weight cycles. How would you modify the algorithm?

*Solution.* Label the vertices  $1, \dots, n$ .

- **Subproblem:** Let  $\delta_k(i, j)$  denote the distance from  $i$  to  $j$  through paths using the intermediary vertices  $\{1, \dots, k\}$ .
- **Recurrence:** To obtain a solution for  $\delta_k(i, j)$ , we either include the  $k$ th vertex or we exclude it from the shortest path.
  - If we exclude the  $k$ th vertex, then we use the first  $(k - 1)$  vertices and the recurrence becomes  $\delta_k(i, j) = \delta_{k-1}(i, j)$ .
  - If we include the  $k$ th vertex, then we find the distance of the shortest path from  $i$  to  $k$  via the intermediary vertices  $\{1, \dots, k - 1\}$  and then find the distance of the shortest path from  $k$  to  $j$ . Now, it could be the case that  $\delta_{k-1}(k, k) < 0$ ; therefore, we will define the following notation  $(\delta_{k-1}(k, k))^*$  to be  $-\infty$  if  $\delta_{k-1}(k, k) < 0$ . In this case, we obtain the recurrence

$$\delta_k(i, j) = \delta_{k-1}(i, k) + (\delta_{k-1}(k, k))^* + \delta_{k-1}(k, j).$$

Thus, the recurrence becomes

$$\delta_k(i, j) = \min \left\{ \delta_{k-1}(i, j), \delta_{k-1}(i, k) + (\delta_{k-1}(k, k))^* + \delta_{k-1}(k, j) \right\}.$$

- **Base case:** If  $(i, j)$  is not an edge, then set  $w(i, j) = \infty$ , where  $w(i, j)$  denotes the weight of the edge from  $i$  to  $j$ . For each vertex  $i \in V$ , we note that  $\delta_0(i, i) = \min\{0, w(i, i)\}$  and  $\delta_0(i, j) = w(i, j)$  if  $i \neq j$ .
- **Order of Computation and Final Solution:** We order the subproblems in increasing order of  $k$ , with the final solution being a grid of all  $O(n^2)$  shortest paths that satisfy the two constraints in the problem.
- **Time Complexity:** For each  $k = 0, \dots, n$ , we solve  $O(n^2)$  many subproblems. Therefore, the overall time complexity is  $O(n^3)$ .

□

**[H] Exercise 28.** Let  $G = (V, E)$  be a connected and undirected graph, where  $|V| = n$  and  $|E| = m$ . Given two vertices  $s, t \in V$ , design an  $O(n^2)$  dynamic programming algorithm to count the number of shortest paths (in the sense of number of edges required to connect  $s$  and  $t$ ) from  $s$  to  $t$ .

*Solution.* We first preprocess a `dist` array which keeps track of the shortest path from  $s$  to every vertex  $u \in V$ . In other words, `dist[u]` is the length of the shortest path from  $s$  to  $u$ , which can be done using breadth-first search. We now begin the dynamic programming.

- **Subproblem:** Let  $\text{NUM}(u)$  denote the number of shortest paths from  $s$  to  $u$ .
- **Recurrence:** To get to vertex  $u$ , we need to find the number of shortest paths that end at the neighbours of  $u$  whose distance is one less than  $u$ . Therefore, we only count along the paths that end with vertex  $v$  which satisfy the two properties:

- $(u, v) \in E$  is an edge in the graph  $G$  (i.e.  $v$  is a neighbouring vertex to  $u$ ).
- $\text{dist}[v] = \text{dist}[u] - 1$ .

Once we find  $v$ , we then count the number of shortest paths from  $s$  to  $v$  and this determines the number of shortest paths to  $u$ . Hence, our recurrence is

$$\text{NUM}(u) = \sum_{\substack{v: (v,u) \in E, \\ \text{dist}[v] = \text{dist}[u] - 1}} \text{NUM}(v).$$

- **Base case:** We hit the base case precisely when we arrive at the starting vertex  $s$ . There is exactly one shortest path from  $s$  to  $s$ ; therefore,  $\text{NUM}(s) = 1$ .
- **Order of Computation and Final Solution:** The final solution is  $\text{NUM}(t)$ . Note that each subproblem relies on information about neighbouring vertices which have a smaller “breadth-first search” distance. Therefore, we solve each subproblem in increasing order of distance computed by the precomputation of breadth-first search.
- **Time complexity:** Even though each subproblem requires at most  $O(n)$  work, observe that we only need to worry about computing the subproblems for each *neighbouring* vertex. Therefore, each subproblem requires  $O(\deg(u))$  work. Hence, the total work required is equivalent to  $\sum_{u \in V} \deg(u)$  amount of work. But this is the same as  $2m$ , by the Handshaking Lemma; therefore, the total running time is  $O(m) = O(n^2)$ .

□

**[H] Exercise 29.** There are  $2n$  students in a class and these students are sitting around a circular table. We form  $n$  pairs of students. When two students are paired up (say student  $i$  and student  $j$ ), we connect a line from student  $i$  to student  $j$ . Design an  $O(n^2)$  algorithm to find the number of ways we can form  $n$  pairs such that no two lines intersect.

Look at small cases of  $n$ . Once we connect two students, can we arbitrarily connect two other students?

*Solution.* The key insight is that, once we draw a line that connects two points on the circle, then this divides our circle into two smaller sets, say  $S_1$  and  $S_2$ . If we connect an edge connecting a point in  $S_1$  and a point in  $S_2$ , then this must intersect the chord that we drew. Therefore, this is equivalent to solving the same problem on the smaller sets,  $S_1$  and  $S_2$ .

- **Subproblem:** Let  $\text{NUM}(i)$  denote the number of ways to pair up  $2i$  points to form  $i$  pairs with non-intersecting lines.
- **Recurrence:** Note that, once we fix a pair, we subdivide into two regions with  $2j$  and  $2(i-j-1)$  points, with  $j = 0, \dots, i-1$ . It, therefore, suffices to find the number of pairs that we can form on both of these separate regions. In the first region with  $2j$  points, we can form  $\text{NUM}(j)$  many pairs while the second region with  $2(i-j-1)$  points can form  $\text{NUM}(i-j-1)$  pairs. We take all possible subdivisions, giving

$$\text{NUM}(i) = \sum_{j=0}^{i-1} \text{NUM}(j) \cdot \text{NUM}(i-j-1).$$

- **Base case:** If there are only two points, then we can only form one pair; therefore,  $\text{NUM}(1) = 1$  and  $\text{NUM}(0) = 0$ .
- **Order of Computation and Final Solution:** We solve this in increasing order of  $i$  with the final solution being  $\text{NUM}(n)$ .
- **Time Complexity:** There are  $n$  subproblems and each subproblem requires at most  $O(n)$  work, giving us an  $O(n^2)$  algorithm.

□

**[E] Exercise 30.** Let  $G = (V, E)$  be a directed graph, where  $|V| = n$  and  $|E| = m$ . Given two vertices  $s, t \in V$ , design an algorithm that determines whether  $s$  and  $t$  are connected. Aim to do this by only using  $O(\log^2 n)$  amount of space. Your algorithm would have exponential running time.

This problem is the basis for an important breakthrough result in computational complexity theory, [Savitch's theorem](#).

*Solution.* We combine dynamic programming with divide and conquer.

- **Subproblem:** Let  $\text{CONNECTED}(u, v, k)$  denote whether  $u, v \in V$  are connected in  $k$  steps.
- **Recurrence:** To do this properly, we will invoke non-determinism. Guess a vertex  $w$ . Then notice that  $\text{CONNECTED}(u, v, k)$  is true if and only if  $\text{CONNECTED}(u, w, k/2)$  and  $\text{CONNECTED}(w, v, k/2)$  are both simultaneously true since we have the path

$$\underbrace{u \rightarrow \cdots \rightarrow w}_{k/2 \text{ steps}} \quad \underbrace{w \rightarrow \cdots \rightarrow v}_{k/2 \text{ steps}}$$

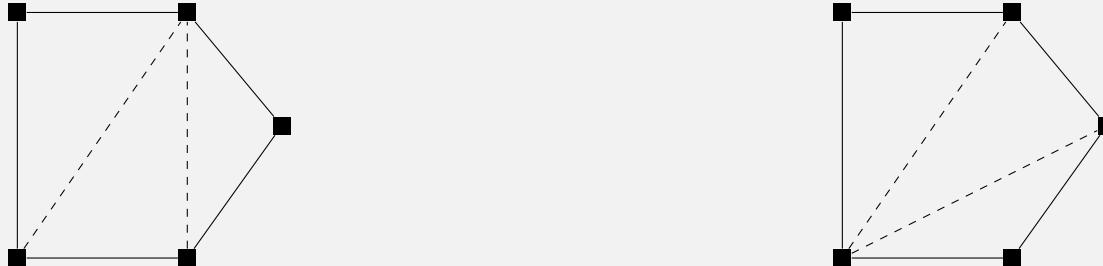
Thus, we have the recurrence

$$\text{CONNECTED}(u, v, k) = \bigvee_{w \in V} (\text{CONNECTED}(u, w, k/2) \wedge \text{CONNECTED}(w, v, k/2))$$

- **Base case:** If  $u = v$ , then  $\text{CONNECTED}(u, v, k) = \text{true}$ . If  $k = 1$ , then we just need to check if there is an edge from  $u$  to  $v$ .
- **Order of Computation and Final Solution:** We solve in increasing order of  $k$ , only storing the vertex  $w$  such that  $\text{CONNECTED}(u, w, k/2)$  and  $\text{CONNECTED}(w, v, k/2)$  are both true. The final solution is  $\text{CONNECTED}(s, t, n)$ .
- **Space Complexity:** At each iteration, we only store the solution vertex  $w$ . Since there are  $\log n$  many iterations, we only require  $O(\log n)$  storage non-deterministically. Savitch's theorem ensures that there is a deterministic solution only requiring  $O(\log^2 n)$  space.

□

**[E] Exercise 31.** A convex polygon is a closed shape where none of its edges bend inwards. Another way to think of this is, if you connect any two points inside the polygon, the line segment remains completely inside the polygon. A *triangulation* of a convex polygon is a deconstruction of the convex polygon into triangles such that the diagonals do not intersect. The following diagram describes possible triangulations of the same convex polygon.



We define the weight of a triangle to be the perimeter of the triangle, and we define the cost of a triangulation to be the sum of the weights of its component triangles. Given  $n$  points and some corresponding graph data structure to define the convex polygon, design an  $O(n^3)$  algorithm to find the minimum cost triangulation of the convex polygon.

*Solution.* Label the vertices  $1, \dots, n$ .

- **Subproblem:** Let  $\text{OPT}(i, j)$  denote the minimum cost polygon triangulation using vertices  $i, \dots, j$ .
- **Recurrence:** Define  $\text{COST}(i, j, k)$  to be the sum of the perimeter of the triangle with vertices  $i, j, k$ . To obtain a suitable recurrence, we look at all possible line segments. Note that each line segment divides our convex polygon into two smaller convex polygons. Therefore, we look at *all* possible ways to divide our polygon into two smaller polygons and form a triangle with the vertices  $i, k, j$  where  $i + 1 \leq k \leq j - 1$ . This gives us a cost of  $\text{COST}(i, k, j)$  and we obtain the polygons  $\text{OPT}(i, k)$  and  $\text{OPT}(k, j)$ . Thus, we obtain

$$\text{OPT}(i, j) = \min_{i < k < j} \{\text{OPT}(i, k) + \text{OPT}(k, j) + \text{COST}(i, k, j)\}.$$

- **Base case:** If  $i = j$ , then we have no cost. Therefore,  $\text{OPT}(i, i) = 0$ .
- **Order of Computation and Final Solution:** We solve these subproblems in increasing order of the size of the polygons; therefore, we solve in increasing order of  $j - i$ . The final solution is  $\text{OPT}(1, n)$ .
- **Time Complexity:** There are  $n^2$  many subproblems and we look at all possible triangulations. This gives us  $O(n)$  work for each pair of indices, giving us an  $O(n^3)$  algorithm.

□

# COMP3121/9101

## ALGORITHM DESIGN

### PRACTICE PROBLEM SET 6 – SELECTED TOPICS

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

## Contents

|  |           |
|--|-----------|
| <b>1 SECTION ONE: STRING MATCHING</b>    | <b>2</b>  |
| <b>2 SECTION TWO: LINEAR PROGRAMMING</b> | <b>6</b>  |
| <b>3 SECTION THREE: INTRACTABILITY</b>   | <b>14</b> |

## § SECTION ONE: STRING MATCHING

**[K] Exercise 1.** Consider the text  $T = \text{"dbebfjcgfdfj"}$  and pattern  $P = \text{"dh"}$ , taken from the alphabet  $\{\text{'a'}, \dots, \text{'j'}\}$  of size  $d = 10$ . We will use the Rabin-Karp algorithm to search for matches of  $P$  in  $T$ , choosing  $p = 11$ .

By close inspection, you will observe that  $P$  does not appear as a substring of  $T$  even once (indeed,  $T$  does not contain any instances of 'h'). However, the Rabin-Karp algorithm sometimes produces false positives due to hash collisions.

How many false positives are there? That is, letting  $T_s = t_s t_{s+1}$  for various starting points  $s$ , how many times do we have that  $H(T_s) = H(P)$  but  $T_s \neq P$ ?

*Solution.* First, convert  $T$  and  $P$  to the equivalent numbers in base 10, namely 314159265359 and 37.

We observe that  $37 \pmod{11} = 4$ . So we are looking for substrings of  $T$  of length 2 which hash to 4 in modulo 11. We obtain

$$\begin{aligned} 31 &\pmod{11} = 9 \\ 14 &\pmod{11} = 3 \\ 41 &\pmod{11} = 8 \\ 15 &\pmod{11} = 4 \\ 59 &\pmod{11} = 4 \\ 92 &\pmod{11} = 4 \\ 26 &\pmod{11} = 4 \\ 65 &\pmod{11} = 10 \\ 53 &\pmod{11} = 9 \\ 35 &\pmod{11} = 2 \\ 59 &\pmod{11} = 4, \end{aligned}$$

so there are five false positives: "bf", "fj", "jc", "cg" and "fj". □

**[K] Exercise 2.** Let  $s = 01011011010$  be a string (of length 11) from the alphabet  $\Sigma = \{0, 1\}$ .

- (a) Compute the transition function for the pattern  $s$ , i.e. find  $\delta(k, a)$  for all  $0 \leq k \leq 11$  and  $a \in \Sigma$ .
- (b) Draw the corresponding state transition diagram. You may omit arrows to state 0.
- (c) Compute the prefix function for the pattern  $s$ , i.e. find  $\pi(k)$  for all  $1 \leq k \leq 11$ .

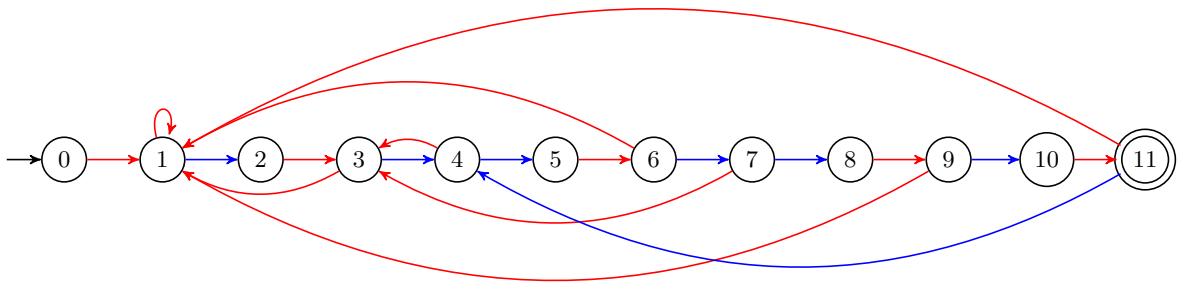
*Solution.* (a) To compute the transition function, we first compute all of the prefixes of  $s$ . We see that the prefixes are

$$\{\epsilon, 0, 01, 010, 0101, 01011, 010110, 0101101, 01011011, 010110110, 0101101101, 01011011010\}.$$

Now, to compute the transition function, we look at the length of the *longest prefix* that can be obtained by appending either a 0 or a 1 at the end of the string that we will have matched. The following transition function is tabulated below.

| Length | String matched | 0  | 1  |
|--------|----------------|----|----|
| 0      | $\epsilon$     | 1  | 0  |
| 1      | 0              | 1  | 2  |
| 2      | 01             | 3  | 0  |
| 3      | 010            | 1  | 4  |
| 4      | 0101           | 3  | 5  |
| 5      | 01011          | 6  | 0  |
| 6      | 010110         | 1  | 7  |
| 7      | 0101101        | 3  | 8  |
| 8      | 01011011       | 9  | 0  |
| 9      | 010110110      | 1  | 10 |
| 10     | 0101101101     | 11 | 0  |
| 11     | 01011011010    | 1  | 4  |

- (b) The diagram is as follows, with red edges representing those labelled 0 and blue edges for those labelled 1.



- (c) The prefix function is shown in the following table.

| $k$      | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----------|---|---|---|---|---|---|---|---|---|----|----|
| $\pi(k)$ | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2  | 3  |

□

**[H] Exercise 3.** Since the Rabin-Karp algorithm functions on a one-dimensional array, explain how you would extend the Rabin-Karp method to look for an  $m \times m$  pattern in an  $n \times n$  array of symbols.

*Solution.* The idea is simple: starting from the first  $m \times m$  grid, we move one column along at a time, row by row, because you need to search through all  $m \times m$  possible grids in the  $n \times n$  grid. For example, let  $T$  be the following.

|          |          |          |
|----------|----------|----------|
| $a_{11}$ | $a_{12}$ | $a_{13}$ |
| $a_{21}$ | $a_{22}$ | $a_{23}$ |
| $a_{31}$ | $a_{32}$ | $a_{33}$ |

$3 \times 3$  grid for  $T$

To find a possible  $2 \times 2$  pattern match, we start with the top left  $2 \times 2$  grid. We, therefore, search for the following four subgrids.

|          |          |               |          |          |               |          |          |               |          |          |
|----------|----------|---------------|----------|----------|---------------|----------|----------|---------------|----------|----------|
| $a_{11}$ | $a_{12}$ | $\Rightarrow$ | $a_{12}$ | $a_{13}$ | $\Rightarrow$ | $a_{21}$ | $a_{22}$ | $\Rightarrow$ | $a_{22}$ | $a_{23}$ |
| $a_{21}$ | $a_{22}$ |               | $a_{22}$ | $a_{23}$ |               | $a_{31}$ | $a_{32}$ |               | $a_{32}$ | $a_{33}$ |

This intuition will allow us to develop a strategy to extend the Rabin-Karp algorithm. To this end, we convert our  $m \times m$  grid into an integer using its  $m^2$  characters. We do this in two parts: the first part, we convert each column into a unique integer; in this way, our  $m$  columns give us a unique integer.

In the second stage, we convert our  $n \times n$  grid into  $m \times m$  blocks. Starting with the top leftmost  $m \times m$  block, we convert this into an integer  $t_{0,0}$ . Then, for each subsequent  $m \times m$  block, we can update the value in constant time by shifting the  $m \times m$  to the right by one column to obtain  $t_{0,1}$ . This is done in  $O(m)$  time because each column of the  $n \times n$  grid correspond to a unique integer which is produced by the  $m$  integers. These are used to update subsequent  $m \times m$  blocks since  $t_{i,j+1}$  can be computed by appealing to the value of  $t_{i,j}$ .

The above construction is used to compute subsequent  $m \times m$  blocks by moving one column to the right. To compute  $m \times m$  blocks by moving one row down, we approach it in a similar fashion. Start by updating the  $m$  column numbers and compute  $t_{i,j}$  for each  $j$ . This can be done since each column number is given by the previous row; hence the construction will take  $O(m)$  time. The construction can be made unique by using some basis  $d$  and the basis numbers  $d^m$  are used to compute all  $t_{i,j}$  column numbers.

The construction takes polynomial time and we have effectively converted the  $n \times n$  case to the one-dimensional Rabin-Karp method. Therefore, to find if the  $m \times m$  grid is a subgrid of  $T$ , we use a suitable prime modulo  $q$  where  $q = \Omega(m^2)$ .  $\square$

**[H] Exercise 4.** Let  $T$  and  $T'$  be strings of length  $n$ . Describe an  $O(n)$  time algorithm to determine if  $T$  and  $T'$  are cyclic rotations of one another. For example,  $T = \text{car}$  and  $T' = \text{arc}$  are cyclic rotations of each other, while  $T = \text{arc}$  and  $T' = \text{acr}$  are not.

*Solution.* If  $T$  and  $T'$  are cyclic rotations of one another, they must have the same number of characters. Therefore, if  $|T| \neq |T'|$ , then they can't be cyclic rotations. Now consider the string  $TT$ . Our claim is that  $T$  and  $T'$  are cyclic rotations of one another if and only if  $T'$  is a substring of  $TT$ .

Indeed, suppose that  $T$  and  $T'$  are cyclic rotations. Then we can think of  $T'$  as shifting  $T$  a finite number (say  $s$ ) of times to the right. In other words, the  $|T| - s$  prefix of  $T$  must be a suffix of  $T'$  and the  $s$  suffix of  $T$  is a prefix of  $T'$ . By concatenating  $T$  with itself and by the characterisation above, it follows that  $T'$  is a substring of  $TT$ .

Suppose that  $T'$  is a substring of  $TT$  with a shift of  $s$ . Then it follows that  $s$  suffix of  $T$  is a prefix of  $T'$ , and  $|T| - s$  characters left in  $T'$  form a prefix of  $T$ . In other words,  $T$  and  $T'$  are cyclic rotations of one another.

Thus, we can use KMP to determine whether  $T'$  is a substring in  $TT$ , which can be done in linear time.  $\square$

**[H] Exercise 5.** Given a list of strings  $A$  and a prefix string  $s$ , describe an algorithm to count the number of strings whose prefix matches  $s$ .

*Solution.* Suppose that  $s$  is of length  $k$ . An intuitive solution is to look for all prefix strings of size  $k$  in  $A$ . If a string is of smaller length, we discard it from consideration since  $s$  cannot be a prefix of such a string. Then, for each prefix string  $s'$  in our array, we check for equality. If equality is met, we increment our counter. Otherwise, we continue. The result will count the number of strings with prefix  $s$ .

Since each string is truncated to a string of size  $k$ , the operation of setting up an auxiliary array (or equivalent) to construct all prefixes of size  $k$  is  $O(k \cdot |A|)$ . Then, for each string in the new array, we check equality in  $O(k)$  time. Thus, we also have  $O(k \cdot |A|)$  time in the second operation.

Alternatively, we can implement a trie data structure. This will allow us to keep track of *all* possible prefixes. By attaching a counter for each trie node, the counter efficiently counts all strings that have a certain string as its prefix. From this construction, we can find the number of strings that have a certain prefix by traversing from the root to the particular substring in the trie data structure, which has length  $k$ . The time complexity is  $O(|A| \cdot \ell)$  where  $\ell$  represents the longest word in  $A$ .  $\square$

**[E] Exercise 6.** Given two patterns  $T$  and  $T'$ , describe how you would construct a finite automaton that determines all occurrences of either  $T$  or  $T'$ .

*Solution.* To give a sketch of the solution, note that we can use KMP to find occurrences of one pattern. Thus, we can employ KMP on  $T$  and  $T'$  separately. This gives us two automata. We can then build a non-deterministic finite automata that accepts the “union” of the two automata. Starting with the first state (i.e. the state that matches strings of length 0), we non-deterministically choose to match either  $T$  or  $T'$ , and follow along the chosen automaton. Each subsequent state transition is determined uniquely by the transition provided by the corresponding automaton given when running KMP on the respective strings.

The final automaton yields all occurrences of either  $T$  or  $T'$ .  $\square$

## § SECTION TWO: LINEAR PROGRAMMING

**[K] Exercise 7.** Sydney Sounds Manufacturing produces speakers for hi-fi sets in two types, product  $A$  and product  $B$ . Two types of processes are needed for their production. The first process combines all machining operations, and the second consists of all assembly operations. Each unit of product  $A$  requires 4 labor-hours of machining and 2 labor-hours of assembly work, whereas each unit of product  $B$  requires 3 labor-hours of machining and 0.5 labor-hours of assembly work. Manufacturing capacity available during the coming production week is 2400 labor-hours, and the assembly work capacity available for the same week is 750 labor-hours.

Previous sales experience indicates that product  $B$  sells at least as much as product  $A$  and that there is already an order of 100 units for product  $A$  to be produced during the next period. Furthermore, all products produced during the week can be sold during the same week. Products  $A$  and  $B$  provide \$7.00 and \$5.00 profit per unit sold, respectively. Management would like to know what quantities of  $A$  and  $B$  should be manufactured during the next production week in order to maximize the total profit. The following table summarizes the data.

| Operation               | Product A | Product B | Capacity |
|-------------------------|-----------|-----------|----------|
| Machining (labor-hours) | 4         | 3         | 2400     |
| Assembly (labor-hours)  | 2         | 0.5       | 750      |
| Profit per unit (\$)    | 7         | 5         |          |

- (a) Write down the linear program  $P$  representing this problem in standard form, making sure to define all variables and constraints involved.
- (b) Write down the dual  $P^*$  of the problem  $P$ .
- (c) Find the optimal value of the objective function and the quantities to produce in order to maximize the total profit.

**Hint.** You can use MATLAB or other software to find the solution, however, there is a way to compute it analytically, maybe draw a plot?

*Solution.* (a) Let us define the vector we wish to find as  $\mathbf{x} = (x_1, x_2)$  with

- $x_1$  = number of unit of product  $A$  produced per week.
- $x_2$  = number of unit of product  $B$  produced per week.

Basing on the constraints, we formulate the optimization problem as follows

$$\begin{array}{ll}
 \text{Maximize}_{\mathbf{x} \in \mathbb{R}^2} & f(\mathbf{x}) = 7x_1 + 5x_2 & \text{maximize profit} \\
 \text{Subject to} & 4x_1 + 3x_2 \leq 2400, & \text{limit on the capacity for machining} \\
 & 2x_1 + \frac{1}{2}x_2 \leq 750 & \text{limit on the capacity for assembly} \\
 & x_1 - x_2 \leq 0 & B \text{ sells at least as much as } A \\
 & -x_1 \leq -100 & \text{existing orders} \\
 & x_1, x_2 \geq 0 & \text{cannot produce negative number of units}
 \end{array}$$

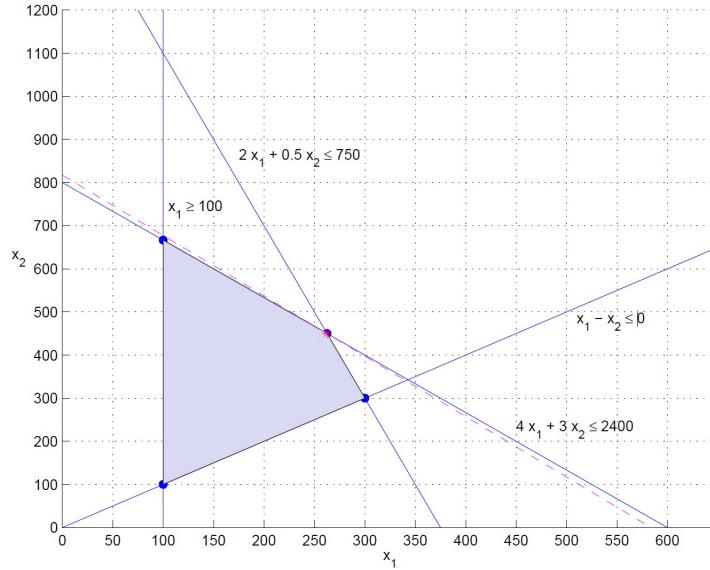
- (b) We can write  $P$  in the triplet form of  $(A, \mathbf{b}, \mathbf{c})$  with

$$\mathbf{c} = \begin{bmatrix} 7 \\ 5 \end{bmatrix} \quad A = \begin{bmatrix} 4 & 3 \\ 2 & \frac{1}{2} \\ 1 & -1 \\ -1 & 0 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 2400 \\ 750 \\ 0 \\ -100 \end{bmatrix} \quad \mathbf{x} \geq 0.$$

Then, we can easily write its dual problem  $P^*$  as

$$\begin{aligned} & \underset{\mathbf{y} \in \mathbb{R}^4}{\text{Minimize}} \quad f^*(\mathbf{y}) = \mathbf{b}^T \mathbf{y} \\ & \text{subject to} \quad A^T \mathbf{y} \geq \mathbf{c} \\ & \quad \text{and} \quad \mathbf{y} \geq 0. \end{aligned}$$

(c) Here we notice that as all the constraints are linear, we can sketch the feasible region of  $\mathbf{x}$  as follows:



Note that for  $P$ , maximizing  $f(\mathbf{x})$  is equivalent to minimizing  $-f(\mathbf{x})$ . Furthermore, we propose the following:

**Claim.** For an optimization problem  $P$  with a convex function  $f(\cdot)$  and convex feasible region  $\Omega$ , the global minimum must exist on an extreme point of  $\Omega$ .

*Solution.* We will omit the proof as it is beyond our scope, but you can read more about it [here](#). □

We can then just simply examine the four points at

$$\mathbf{x} = \begin{pmatrix} 100 \\ 100 \end{pmatrix}, \quad \begin{pmatrix} 300 \\ 300 \end{pmatrix}, \quad \begin{pmatrix} 262.5 \\ 450 \end{pmatrix}, \quad \begin{pmatrix} 100 \\ \frac{2000}{3} \end{pmatrix}.$$

Then in this case our optimal solution yields

$$\mathbf{x}^* = \begin{pmatrix} 262.5 \\ 450 \end{pmatrix} \quad f(\mathbf{x}^*) = 4087.5.$$

□

**[K] Exercise 8.** Suppose that there are 3 farmers each with a square of land with the side length of  $s_i$  and center of the land at  $(x_i, y_i)$ . However, due to the current drought, the farmers need to drill a single well to extract water and keep the crops alive. The well's position must be chosen such that it is within the land of all 3 farmers. The amount of water that is extractable from a coordinate  $(x, y)$  can be written as  $w = 4x + 6y$  and you may assume that no land will include negative coordinates.

- (a) Is there always a valid solution for the position of the well? Explain your answer.

- (b) Suppose a valid solution exists, derive the standard form LP formulation  $P$  that finds the correct position to drill the well such that most amount of water can be extracted while satisfying the requirement.
- (c) How can you extend the definition of this problem to  $n$  farmers?

*Solution.* (a) There exists no solution if there exists there are no overlap between the lands of any pair of farmers. Formally, let  $r_i = s_i/2$ , if exists farmer  $i$  and farmer  $j$  such that for

$$\begin{aligned} O_{\text{row}} &= (x_i - r_i \leq x_j + r_j) \wedge (x_j - r_j \leq x_i + r_i) \\ O_{\text{col}} &= (y_i - r_i \leq y_j + r_j) \wedge (y_j - r_j \leq y_i + r_i) \\ O^* &= O_{\text{row}} \wedge O_{\text{col}}. \end{aligned}$$

we do not have  $O^* = T$ , then there is no suitable position for the well.

- (b) We start by noting that our constraint requires that

$$|x - x_i| \leq r_i \quad |y - y_i| \leq r_i \quad \text{for } i = 1, 2, 3$$

hence  $(x, y)$  must be within the field of each farmer. However this is not in standard form, we need to transform each of the clauses to

$$|x - x_i| \leq r_i \implies x - x_i \leq r_i \quad \text{and} \quad -x + x_i \leq r_i$$

Therefore, we can write the standard form LP  $P$  for  $\mathbf{x} = (x, y)$ ,

$$\begin{array}{ll} \underset{\mathbf{x} \in \mathbb{R}^2}{\text{Maximize}} & f(\mathbf{x}) = 4x + 6y \\ & \text{maximize } w \text{ the amount of water} \\ \text{Subject to} & x - x_i \leq r_i \quad \text{for } i = 1, 2, 3, \\ & y - y_i \leq r_i \quad \text{for } i = 1, 2, 3, \\ & -x + x_i \leq r_i \quad \text{for } i = 1, 2, 3, \\ & -y + y_i \leq r_i \quad \text{for } i = 1, 2, 3, \\ & x, y \geq 0 \quad \text{no negative coordinates.} \end{array}$$

We can then write down the triplet form of  $(\mathbf{c}, A, \mathbf{b})$  with

$$\mathbf{c} = \begin{pmatrix} 4 \\ 6 \end{pmatrix} \quad A = \begin{pmatrix} I_{2 \times 2} \\ I_{2 \times 2} \\ I_{2 \times 2} \\ -I_{2 \times 2} \\ -I_{2 \times 2} \\ -I_{2 \times 2} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} r_1 + x_1 \\ r_1 + y_1 \\ \vdots \\ r_3 + x_3 \\ r_3 + y_3 \\ r_1 - x_1 \\ r_1 - y_1 \\ \vdots \\ r_3 - x_3 \\ r_3 - y_3 \end{pmatrix}.$$

With  $I_{k \times k}$  as the identity matrix of size  $k$ .

**Hint.** Is there a way to compute the feasible region directly?

- (c) Natural extension to  $n$  farmers will require

- Our condition are satisfied for (a).

- Our LP problem then becomes

$$\begin{aligned} \text{Maximize}_{\mathbf{x} \in \mathbb{R}^2} \quad & f(\mathbf{x}) = 4x + 6y \\ \text{Subject to} \quad & x - x_i \leq r_i \quad \text{for } i = 1, \dots, n, \\ & y - y_i \leq r_i \quad \text{for } i = 1, \dots, n, \\ & -x + x_i \leq r_i \quad \text{for } i = 1, \dots, n, \\ & -y + y_i \leq r_i \quad \text{for } i = 1, \dots, n, \\ & x, y \geq 0 \end{aligned}$$

Our matrix also changes to

$$A = \left( \underbrace{I_{2 \times 2}, \dots, I_{2 \times 2}}_{n \text{ times}}, \underbrace{-I_{2 \times 2}, \dots, -I_{2 \times 2}}_{n \text{ times}} \right)^T \quad \mathbf{b} = \begin{pmatrix} r_1 + x_1 \\ r_1 + y_1 \\ \vdots \\ r_n + x_n \\ r_n + y_n \\ r_1 - x_1 \\ r_1 - y_1 \\ \vdots \\ r_n - x_n \\ r_n - y_n \end{pmatrix}.$$

as required. □

**[K] Exercise 9.** You are the boss of a large manufacturing company and you wish to produce a certain amount of items to sell to the customers while making as much profit as possible.

- Based on the technical constraint, you can choose to produce any amount of  $n$  different types of items each with a cost of  $c_i$ .
  - Your accounting department also gives an estimate of the profit for each unit of item  $i$  is  $p_i$  and your budget is  $C$  in total.
  - Manufacturing each item  $i$  also produces a certain amount of pollution. Based on the constraint set by EPA, there are  $k$  many different types of pollution measures and limit you must adhere to for each pollution measure be denoted by  $L_i$ .
  - Producing each item  $i$  will cause any amount of pollution for each of the different measures, we denote this by  $(w_{(i,1)}, w_{(i,2)}, \dots, w_{(i,k)})$ .
- (a) Classify this problem as either Linear Programming or Integer Linear Programming, and deduce whether there is a known algorithm to solve it in polynomial time?
  - (b) Formulate this problem  $P$  in standard form, making sure to define all variables and constraints involved.
  - (c) Formulate the dual of this problem  $P^*$ .
  - (d) Suppose that all of a sudden, exactly  $m < n$  many types items are required to produce exactly  $\nu_1, \nu_2, \dots, \nu_m$  many. How should you modify your LP formulation? (you can assume that producing those items do not exceed the pollution requirement)

*Solution.* (a) Since we cannot produce half an item, this problem is an **Integer Linear programming** problem, hence there is no known polynomial time algorithm for solving it.

(b) We start by defining our variables, let  $x_i$  denote the number of units of item type  $i$ . Then with  $\mathbf{x} = (x_1, \dots, x_n)$  We can formulate our standard form LP  $P$  as

$$\begin{array}{ll} \text{Maximize}_{\mathbf{x} \in \mathbb{Z}^n} & f(\mathbf{x}) = \sum_{i=1}^n x_i p_i & \text{maximize the profit obtainable} \\ \text{Subject to} & \sum_{i=1}^n c_i x_i \leq C & \text{no spending exceeds } C \\ & \sum_{i=1}^n w_{(i,j)} x_i \leq L_j \quad \text{for } j = 1, \dots, k & \text{cannot exceed total pollution limit} \\ & \mathbf{x} \geq 0 & \text{cannot manufacture negative units} \end{array}$$

Then we can obtain its matrix triplet,  $(\mathbf{c}, A, \mathbf{b})$  namely

$$\mathbf{c} = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad A = \begin{pmatrix} c_1 & c_2 & c_3 & \dots & c_n \\ w_{(1,1)} & w_{(2,1)} & w_{(3,1)} & \dots & w_{(n,1)} \\ w_{(1,2)} & w_{(2,2)} & w_{(3,2)} & \dots & w_{(n,2)} \\ w_{(1,3)} & w_{(2,3)} & w_{(3,3)} & \dots & w_{(n,3)} \\ \vdots & \dots & \dots & \ddots & \vdots \\ w_{(1,k)} & w_{(2,k)} & w_{(3,k)} & \dots & w_{(n,k)} \end{pmatrix} = \begin{pmatrix} \gamma_{1,n} \\ W_{k,n} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} C \\ L_1 \\ L_2 \\ L_3 \\ \vdots \\ L_k \end{pmatrix}$$

(c) From the matrix triplet we have derived above, the dual of  $P$  is then

$$\begin{array}{ll} \text{Minimize}_{\mathbf{y} \in \mathbb{Z}^{k+1}} & f(\mathbf{y}) = \mathbf{b}^T \mathbf{y} \\ \text{Subject to} & A^T \mathbf{y} \geq \mathbf{c} \\ & \mathbf{y} \geq 0 \end{array}$$

(d) Here we modify our  $P$  such that it adheres to the equality constraint. Without loss of generality, let  $1, \dots, m$  denote the items types that requires exact units of production and let  $m+1, \dots, n$  be the unrestricted types. We can then reduce our optimization problem  $P'$  with  $\mathbf{z} = (z_1, \dots, z_{n-m})$

$$\begin{array}{ll} \text{Maximize}_{\mathbf{z} \in \mathbb{Z}^{n-m}} & f(\mathbf{z}) = \sum_{i=1}^{n-m} p_{m+i} z_i \\ \text{Subject to} & \sum_{i=1}^{n-m} z_i c_{m+i} \leq C - \sum_{i=1}^m c_i \nu_i, \\ & \sum_{i=1}^{n-m} w_{(m+i,j)} z_i \leq L_j - \sum_{i=1}^m w_{(i,j)} \nu_i \quad \text{for } j = 1, \dots, k, \\ & \mathbf{z} \geq 0. \end{array}$$

Or equivalently, we can also modify our problem by adding  $2m$  many restrictions such that  $\mathbf{x}_i \leq \nu_i$  and  $\mathbf{x}_i \geq \nu_i$  for  $i = 1, \dots, m$  to enforce the equality constraint. The matrix form of  $P'$  will be *left as an exercise for the reader*.

□

**[H] Exercise 10.** Linear programming comes up in financial mathematics, here for this problem we will give a preview of the problem of arbitrage betting. Say there is a huge sports tournament going on with  $m$  teams competing and  $n$  different betting agencies currently allowing you to place bets on the outcome of the tournament.

- Suppose that each betting agency  $i$  now allows you to put a bet on the  $t_i$ th team wins at the end of the tournament with a payout of  $f_i : 1$ , write down a standard form (matrix form) LP problem  $P$  that allows you to maximize your risk-free profit (i.e., a strategy that yields a profit regardless of the outcomes) with a budget of  $B$ .
- Write down the dual problem  $P^*$  for the LP formulation you have derived in (a).
- Does your LP procedure always give a solution that will make you money? Explain your answer.

**Hint.** You may want to consider the [Arbitrage Theorem](#).

*Solution.* (a) We start by defining our variables, let  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  be the amount placed at the  $n$ th betting agency. We then define another set of variables  $r$  which

$$r_{(i,j)} = \begin{cases} f_i - 1 & \text{if } j = t_i \\ -1 & \text{otherwise} \end{cases}$$

which is the return for a bet of \$1 on agency  $i$  if the outcome is  $j$ . We now define the concept of a composite return for betting on a specific outcome  $j$  as

$$\varphi_j = \sum_{i=1}^n x_i r_{(i,j)}.$$

therefore, a risk free strategy will require that  $\varphi_j > 0$  for all  $j = 1, \dots, m$ . However, we will also introduce  $x_{n+1}$  as the minimal profit for each outcome. We can then formulate our LP problem  $P$ ,

|   |  |                                      |
|---|--|--------------------------------------|
| Maximize<br>$\mathbf{x} \in \mathbb{R}^{n+1}$ | $f(\mathbf{x}) = x_{n+1}$                                  | maximize minimal profit              |
| Subject to                                    | $\varphi_j \geq x_{n+1} \quad \text{for } j = 1, \dots, m$ | optimize the profit for each outcome |
|   | $\sum_{i=1}^n x_i \leq B$                                  | cannot spend over budget             |
|   | $\mathbf{x} \geq 0$  | cannot give betting agency money     |

Or equivalently, we can write this problem in the triplet form via

$$\mathbf{c} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \quad A = \begin{pmatrix} -r_{(1,1)} & -r_{(2,1)} & \dots & -r_{(n,1)} & 1 \\ -r_{(1,2)} & -r_{(2,2)} & \dots & -r_{(n,2)} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -r_{(1,m)} & -r_{(2,m)} & \dots & -r_{(n,m)} & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ B \end{pmatrix}.$$

- The dual  $P^*$  can again be derived directly from the matrix form via

|   |   |  |
|---|---|--|
| Minimize<br>$\mathbf{y} \in \mathbb{R}^{m+1}$ | $f(\mathbf{y}) = \mathbf{b}^T \mathbf{y}$ |  |
| Subject to                                    | $A^T \mathbf{y} \geq \mathbf{c}$          |  |
|   | $\mathbf{y} \leq 0$                       |  |

(c) We can check the conditions outlined by a simplified version of the Arbitrage Theorem.

**Theorem** (Arbitrage Theorem). For the discrete probability  $p_1, p_2, \dots, p_m$  for the outcome of the tournament, we have that either

- (1) there is a betting strategy  $\mathbf{x}$  which

$$\sum_{i=1}^n x_i r_{(i,j)} > 0 \quad \text{for } j = 1, \dots, m$$

- (2) or we have that

$$\sum_{j=1}^m p_j r_{(i,j)} = 0 \quad \text{for } i = 1, \dots, n$$

So (2) clause tells us that no matter how we bet, our expected return is always 0. □

**[H] Exercise 11.** Linear programming shows up in game theory as well. Suppose there are  $n$  armies advancing on  $m$  cities and each army  $i$  is commanded by a General  $G_i$  with  $R_i$  many regiments.

- Each general  $G_i$  can send choose to send some amount of regiments to a city.
- In each city, the army that sends the most amount of regiments to the city captures both the city and **all other** army's regiments.
- If there are any ties in the number of regiment for the 1st place, then those corresponding armies draws and loses no points. The rest of the armies are penalised the same amount of points as the regiments sent to that city.
- Each army scores 1 point per city captured and 1 point per captured regiment.

Assume that each army needs to make full use of its regiments, and wants to maximize the sum of the difference between its reward and all other opponent's reward.

- (a) This is an example of a zero-sum game. Denote a strategy  $\pi_j = (\pi_1, \pi_2, \dots, \pi_m)$  as follows: the general  $j$  sends  $\pi_i$  regiments to city  $i$  and so on and let  $\mathcal{S}_i$  denote the space of all possible strategies that can be done by general  $i$ .

We define the *payoff matrix (or tensor)*  $C_k : \mathcal{S}_1 \times \mathcal{S}_2 \times \dots \times \mathcal{S}_n \rightarrow \mathbb{Z}$ , which each of the  $C(\pi_1, \pi_2, \dots, \pi_n)$  denotes the payoff of each of the generals selecting the strategy of  $\pi_1, \pi_2, \dots, \pi_n$  in the perspective of  $G_k$ . E.g. If General 1 uses the strategy  $(4, 0)$  and General 2 uses the strategy  $(3, 0)$  then the corresponding  $C_2((4, 0), (3, 0))$  is  $-4$  as General A captures 1 city and 3 regiments, resulting in a *loss* of  $-4$ .

Generate the payoff matrix  $C_2$  for  $n = 2, m = 2$  and  $R_1 = 4, R_2 = 3$ .

- (b) For  $n = 2$ , write down the standard form LP formulation that finds the optimal strategy for any General in a probabilistic sense.

*Solution.* (a) We start by computing all possible strategies of both generals

- $G_1$ 's strategies are:  $(4, 0), (0, 4), (3, 1), (1, 3), (2, 2)$
- $G_2$ 's strategies are:  $(3, 0), (0, 3), (2, 1), (1, 2)$

Then the payoff matrix is:

|        | (3,0) | (0,3) | (2, 1) | (1, 2) |
|--------|-------|-------|--------|--------|
| (4, 0) | -4    | 0     | -2     | -1     |
| (0, 4) | 0     | -4    | -1     | -2     |
| (3, 1) | -1    | 1     | -3     | 0      |
| (1, 3) | 1     | -1    | 0      | -3     |
| (2, 2) | 2     | 2     | -2     | -2     |

(b) We start by defining our variables,

- let  $\mathbf{y} = (y_1, \dots, y_M)^T \in \mathbb{R}^5$ , with  $y_i$  be the probability that  $G_1$  chooses the strategy specified by row  $i$
- let  $\mathbf{x} = (x_1, \dots, x_N)^T \in \mathbb{R}^4$ , with  $x_i$  be the probability that  $G_2$  chooses the strategy specified by column  $i$

Let  $P$  be the payoff matrix we generated similarly to (a), we can solve this problem via the **Weak Duality Theorem!!** Let us define a standard form of  $P(f, \mathbf{y})$  as

$$\begin{array}{ll} \text{Maximize}_{\mathbf{x} \in \mathbb{R}^N} & f(\mathbf{x}) = \mathbf{y}^T P \mathbf{x} & \text{maximize score in perspective of } G_2 \\ \text{Subject to} & \sum_{i=1}^N x_i = 1 & \mathbf{x} \text{ is a probability distribution} \\ & \mathbf{x} \geq 0 & \text{probability is positive} \end{array}$$

Let us define a standard form of  $Q(f, \mathbf{x})$  as

$$\begin{array}{ll} \text{Minimize}_{\mathbf{y} \in \mathbb{R}^M} & f(\mathbf{y}) = \mathbf{y}^T P \mathbf{x} & \text{minimize the score of } G_1 \\ \text{Subject to} & \sum_{i=1}^M y_i = 1 & \mathbf{y} \text{ is a probability distribution} \\ & \mathbf{y} \geq 0 & \text{probability is positive} \end{array}$$

We can then solve this problem directly via Weak duality between  $P$  and  $Q$ . However, we can actually simplify this further, note that  $\mathbf{y}$  must be a proper discrete probability distribution, we can expand  $f$  and obtain the LP form as below  $P^*$

$$\begin{array}{ll} \text{Maximize}_{\mathbf{x} \in \mathbb{R}^N, r \in \mathbb{R}} & f(\mathbf{x}) = r \\ \text{Subject to} & (P\mathbf{x})^T \mathbf{e}_i \geq r \quad \text{for } i = 1, \dots, N \\ & \sum_{j=1}^N x_j = 1 \\ & \mathbf{x} \geq 0 \end{array}$$

With  $\mathbf{e}_i$  as the  $i$ th vector in standard basis in  $\mathbb{R}^N$ . For optimal solution for  $G_1$  we can replicate the above process but in reverse.

□

## § SECTION THREE: INTRACTABILITY

[K] **Exercise 12.** Determine for each of the following whether it is a polynomial time algorithm.

- (a)  $O(n)$  input,  $O(n \log n)$  running time.
- (b)  $O(n + \log C)$  input,  $O(nC)$  running time.
- (c)  $O(n)$  input,  $O(n^3)$  running time.
- (d)  $O(n)$  input,  $O(2^n)$  running time.

*Solution.* (a) Yes, the running time is bounded by  $n^2$ .

- (b) No, the running time is not bounded by any polynomial in  $n$  and  $\log C$ , since  $C$  is exponential in  $\log C$ .
- (c) Yes, the running time is bounded by  $n^3$ .
- (d) No, the running time is not bounded by any polynomial in  $n$ .

□

[K] **Exercise 13.** A *clique* of a graph  $G = (V, E)$  is a subset  $U \subseteq V$  of vertices such that any two elements,  $u, v \in U$  are adjacent; that is, for any distinct pairs of vertices  $u, v \in U$ , there exist an edge such that  $(u, v) \in E$  of  $G$ .

Consider the optimisation version of the *clique* problem as stated below.

### Maximum Clique

**Instance:** An undirected graph  $G = (V, E)$ .

**Task:** Choose a subset  $U \subseteq V$  of vertices of maximum size such that for any two vertices  $u, v \in U$ , we have  $(u, v) \in E$ .

- (a) Convert this optimisation problem to the corresponding decision problem (Clique).
- (b) Explain how an algorithm which solves the Clique problem can be extended to solve the original optimisation problem (Maximum Clique) with  $\log |V|$  overhead.
- (c) Show that the Clique problem is in class **NP**.

*Solution.* (a) The decision problem is

### Clique

**Instance:** An undirected graph  $G = (V, E)$  and an integer  $k$ .

**Task:** Choose a subset  $U \subseteq V$  consisting of at least  $k$  vertices (if one exists) such that for any two vertices  $u, v \in U$ , we have  $(u, v) \in E$ .

- (b) Binary search for the largest  $k$  for which a clique of size  $\geq k$  exists.
- (c) The certificate is a claimed clique  $U$ . The certifier problem is

### Clique Certifier

**Instance:** An undirected graph  $G = (V, E)$ , an integer  $k$ , and a subset  $U \subseteq V$  of the vertices.

**Task:** Determine whether  $U$  contains at least  $k$  vertices and whether for any two vertices  $u, v \in U$  we have  $(u, v) \in E$ .

This is solved in  $O(n^2)$  by first making an adjacency matrix to represent  $G$ , then counting the vertices of  $U$  and finally checking whether an edge appears between each pair of vertices of  $U$ . The runtime is clearly polynomial in the length of the input ( $O(n + m)$ ). □

[K] **Exercise 14.** Recall the Integer Knapsack problem from the Dynamic Programming lecture. The problem is stated below.

### Integer Knapsack

**Instance:** a positive integer  $n$ , integers  $w_i$  and  $v_i$  for each  $1 \leq i \leq n$ , and a positive integer  $C$ .

**Task:** Choose a combination of items (with repetition allowed) which all fit in the knapsack and whose total value is as large as possible.

- Is this an optimisation or decision problem?
- If this is an optimisation problem, convert it to a corresponding decision problem.
- Show that the decision problem is in **NP**.

*Solution.* (a) This is an optimisation problem, since we are maximising the total value.

- (b) The corresponding decision problem is as follows.

### Integer Knapsack (Decision)

**Instance:** a positive integer  $n$ , integers  $w_i$  and  $v_i$  for each  $1 \leq i \leq n$ , and positive integers  $C$  and  $V$ .

**Task:** Determine whether there is a combination of items (with repetition allowed) which all fit in the knapsack and whose total value is at least  $V$ .

- (c) Consider the certificate  $\mathbf{a} = (a_1, a_2, \dots, a_n)$ , corresponding to picking the  $i$ th item  $a_i$  many times. The certifier problem is then to check whether such a collection has total weight

$$\sum_{i=1}^n a_i w_i \leq C$$

and total value

$$\sum_{i=1}^n a_i v_i \geq V,$$

both of which can be determined in  $O(n)$ . Thus Integer Knapsack (Decision) is in **NP**. □

[K] **Exercise 15.** Suppose that  $U$  and  $V$  are decision problems in class **NP**, and that there exists a polynomial reduction  $f$  from  $U$  to  $V$ . What can you deduce if:

- (a)  $V$  is in class **P**?
- (b)  $V$  is in class **NP-C**?
- (c)  $U$  is in class **P**?
- (d)  $U$  is in class **NP-C**?

*Solution.* (a) If  $V$  is in the class **P**, there is a polynomial time algorithm to solve instances of  $V$ . Now, for an instance  $x$  of problem  $U$ , first find the corresponding instance  $f(x)$  of problem  $V$ , then solve it. This algorithm is also polynomial time, so  $U$  is in class **P**.

- (b) If  $V$  is in class **NP-C**, then by definition any problem in **NP** is polynomially reducible to  $V$ . Therefore, we cannot deduce anything further about  $U$ .
- (c) We cannot deduce anything further about  $V$  from the assumption that  $U$  is in class **P**.
- (d) If  $U$  is in class **NP-C**, then for any **NP** problem  $W$  there is a polynomial reduction  $g$  from  $W$  to  $U$ . By composition, we obtain the polynomial reduction  $f \circ g$  from  $W$  to  $V$ , so  $V$  is in **NP-C** also.

□

[H] **Exercise 16.** Recall that the Vertex Cover problem is as follows:

### Vertex Cover (VC)

**Instance:**  $G = (V, E)$ , an undirected and unweighted graph, and a positive integer  $k$ .

**Task:** Is it possible to choose  $k$  vertices so that every edge is incident to at least one of the chosen vertices?

In lectures, we showed that the Vertex Cover problem is in **NP**-complete. We will use this problem to prove that a related problem is also **NP**-complete.

Consider the Feedback Vertex Set problem stated below.

### Feedback Vertex Set (FVS)

**Instance:**  $G = (V, E)$ , an undirected (or directed) graph, and a positive integer  $k$ .

**Task:** Is it possible to choose  $k \leq |V|$  vertices so that, if we remove all  $k$  vertices and their corresponding adjacent edges from  $G$ , the new graph is cycle-free?

- (a) Show that Feedback Vertex Set is in **NP**.
- (b) We now prove that Feedback Vertex Set is in **NP-hard**. Consider the following reduction.

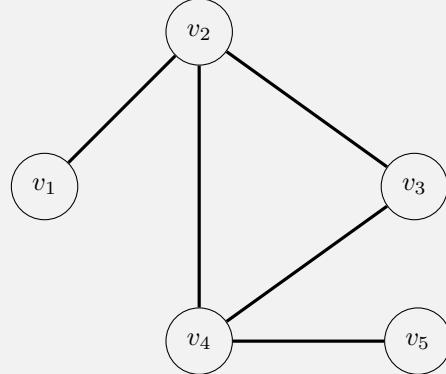
Given a graph  $G = (V, E)$ , we construct  $G' = (V', E')$  where:

- $V' = U_V \cup U_E$ , where  $U_V$  consists of the vertices of  $G$  and, for each edge in  $(v_1, v_2) \in E$ , we create a new vertex  $u_{v_1, v_2}$  which belong in  $U_E$ .
- $E'$  consists of the edges constructed as follows:
  - For each edge  $(v_1, v_2) \in E$  in the original graph, we construct three edges: an edge from  $v_1 \in U_V$  to  $v_2 \in U_V$ , an edge from  $v_1 \in U_V$  to  $u_{v_1, v_2} \in U_E$ , and an edge from  $v_2 \in U_V$  to  $u_{v_1, v_2} \in U_E$ .

We claim that

$$(G, k) \in \text{VC} \iff (G', k) \in \text{FVS}.$$

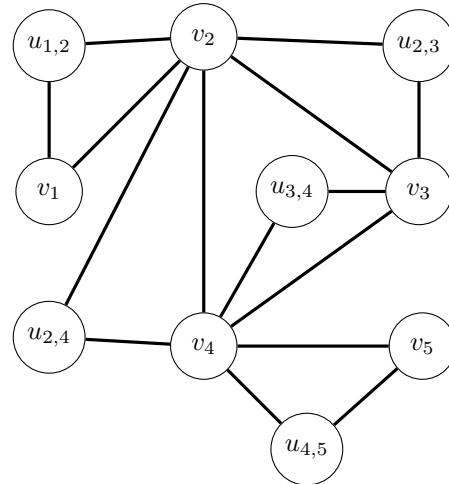
Consider the following graph  $G$ .



- (i) Using the reduction described above, construct  $G'$ .
- (ii) Show that  $G$  has a vertex cover of size 2, and find the corresponding feedback vertex set of size 2 in  $G'$ . How can we generalise this to arbitrary graphs  $G$ , and thus, deduce that Feedback Vertex Set is in **NP**-hard?
- (c) Using the previous two results, conclude that Feedback Vertex Set is in **NP**-complete.

*Solution.* (a) To show that Feedback Vertex Set is in **NP**, we construct a certificate and a polynomial time verifier. The certificate is the subset of  $k$  vertices. To verify whether this is a feedback vertex set, we remove these vertices from  $G$  to construct  $G'$ . Then run a depth-first search on  $G'$  to determine if there are cycles. Since  $G$  is a graph with a finite number of vertices and edges, so is  $G'$ ; the depth-first search takes polynomial time on  $G'$ . Thus, Feedback Vertex Set is in **NP**.

- (b) (i) We construct the following graph,  $G'$ .



- (ii) In  $G$ , we note that the set  $S = \{v_2, v_4\}$  is a vertex cover of size 2 because every edge is incident to some vertex in  $S$ . This is easy to check. In  $G'$ , note that the cycles we construct are caused by edges that were present in the original graph. This tells us that, if we had a vertex cover set in  $G$ , then the corresponding vertices in  $S$  also correspond to the vertices of the Feedback Vertex Set of  $G'$ . In other words, the corresponding

feedback vertex set is  $S' = \{v_2, v_4\}$ .

To see how we generalise this result to arbitrary graphs, we prove that the reduction is valid. In this course, we give a high level reasoning for why  $(G, k) \in \text{VC} \iff (G', k) \in \text{FVS}$ .

The reduction works by forming small 3-cycles,  $\{v_i, u_{i,j}, v_j\}$ , that correspond *precisely* to the edge  $(v_i, v_j) \in E$  in the original graph,  $G$ . If we had a vertex cover  $S$  in  $G$ , then such an edge must be covered by either  $v_i$  or  $v_j$  belonging to  $S$ . Thus, to break the cycle, we need to choose at least one of  $v_i$ ,  $v_j$ , or  $u_{i,j}$ . By choosing the vertex belonging to our vertex cover, we ensure that the 3-cycle will be broken if we remove such a vertex in  $G'$ . In other words, breaking cycles of  $G'$  is equivalent to choosing the vertices that cover every edge of  $G$ .

This shows that the Feedback Vertex Set is as hard as solving Vertex Cover. In other words, Feedback Vertex Set is in **NP-hard**.

- (c) Since Feedback Vertex Set is an **NP** problem (from part (a)) and can be reduced from an **NP**-complete problem (from part (b)), Feedback Vertex Set is consequently in **NP**-complete.

□

**[E] Exercise 17.** Recall that a decision problem  $X$  is said to be in **NP** if *yes* instances of  $X$  have a certificate and a polynomial-time verifier.

In a similar way, we can then define the following class of problems:

**coNP**

A decision problem  $X$  is said to be in **coNP** if *no* instances of  $X$  have a certificate and a polynomial-time verifier.

We say that the *complement* of a problem is the result of switching the “yes” and “no” results. Using the definition of the complement of a problem, we can also define the following class of problems:

**coP**

A decision problem  $X$  is said to be in **coP** if the complement of  $X$  is solvable in polynomial time.

- (a) Show that  $\mathbf{P} = \mathbf{coP}$ .
- (b) Using part (a), show that, if  $\mathbf{P} = \mathbf{NP}$ , then  $\mathbf{NP} = \mathbf{coNP}$ .

*Solution.* (a) We first show that  $\mathbf{P} \subseteq \mathbf{coP}$ . Consider any problem  $X \in \mathbf{P}$ . Then  $X$  can be solved in polynomial time.

In other words, there exist a polynomial running time algorithm that solves (or decides) “yes” and “no” instances of  $X$ . However, just by switching the “yes” and “no” instances of the algorithm gives you a polynomial time algorithm that solves the complement of  $X$ . In other words, if  $X \in \mathbf{P}$ , then  $\bar{X} \in \mathbf{P}$  or equivalently,  $X \in \mathbf{coP}$ . This shows that  $\mathbf{P} \subseteq \mathbf{coP}$ .

The reverse argument shows that  $\mathbf{coP} \subseteq \mathbf{P}$ . This shows that  $\mathbf{P} = \mathbf{coP}$ . The formal argument requires an introduction to Turing machines; however, this is a fairly informal discussion on why  $\mathbf{P} = \mathbf{coP}$ .

- (b) Assuming that  $\mathbf{P} = \mathbf{NP}$ , we have the following

$$\begin{aligned} \mathbf{NP} &= \mathbf{P} && \text{(by assumption)} \\ &= \mathbf{coP} && \text{(from part (a))} \\ &= \mathbf{coNP}. && \text{(since } \mathbf{P} = \mathbf{NP} \text{)} \end{aligned}$$

□

# Solutions to Section II

COMP3121/9101 22T1

Final Exam, 2nd May 2022

This document provides solutions for Section II of the 22T1 final exam. These solutions were prepared primarily as an aid to markers rather than to model full mark solutions for students, so they are a bit rough in places.

## Question 9

You are given a positive integer  $n$ , an array  $A$  of  $n$  positive integers and a positive integer  $S$ . For each pair of indices  $1 \leq i \leq j \leq n$ , consider the subarray  $A[i..j]$ .

Design an algorithm which determines the number of these subarrays with sum strictly less than  $S$  and runs in  $O(n \log n)$  time. You must provide reasoning to justify the correctness and time complexity of your algorithm.

The input consists of the positive integers  $n$  and  $S$ , as well as  $n$  positive integers  $A[1], \dots, A[n]$ .

The output is the number of pairs  $(i, j)$  such that  $A[i] + \dots + A[j] < S$  where  $1 \leq i \leq j \leq n$ .

For example, suppose  $n = 4$ ,  $S = 8$  and the array is  $\langle 2, 5, 3, 4 \rangle$ . There are ten nonempty subarrays, and six of these have sum strictly less than eight, so the answer is six.

## Solution

First, construct an array  $B$  of size  $n$ . Set  $B[1] = A[1]$ , and for all  $2 \leq i \leq n$ , let  $B[i] = B[i - 1] + A[i]$ .  $B[0]$  will be understood as zero. Now, each term  $B[i]$  stores the sum  $A[1] + \dots + A[i]$ , so we can look up the sum of the subarray  $A[i..j]$  in constant time by evaluating  $B[j] - B[i - 1]$ .

Now, we return to the original problem, which is to find the number of pairs  $(i, j)$  such that  $B[j] - B[i - 1] < S$ . For a fixed  $i$ , this amounts to finding the number of indices  $j$  such that  $B[j] < S + B[i - 1]$ . However, the array  $B$  is increasing, so we can find the largest such  $j$  using binary search and add  $j - i + 1$  to the answer. Repeating this for all  $i$ , we find the number of pairs.

Calculating array  $B$  takes linear time, and for each index  $i$  the binary search takes  $O(\log n)$  time, so the total time complexity is  $O(n \log n)$ .

## Question 10

You are given an  $n$  digit positive integer  $A$  and a positive integer  $k < n$ . Let the digits of  $A$  be  $a_{n-1}, a_{n-2}, \dots, a_0$ , in order from most to least significant. You can delete  $k$  digits from  $A$  to leave an  $(n - k)$  digit number  $B$ , potentially with leading zeroes.

Design an algorithm which determines the smallest possible value  $B$  and runs in  $O(n)$  time.

The input consists of the positive integers  $n$ ,  $A$  and  $k$ .

The output is an  $(n - k)$  digit positive integer, which may include leading zeroes.

For example, if  $n = 5$ ,  $A = 90834$  and  $k = 2$ , the correct answer is  $B = 034$ .

## Solution

We loop from the most significant digit to the least significant digit and maintain a stack which is initially empty.

For each digit  $a_i$ , while it is smaller than the last element on the stack, then keep popping off from the top of the stack (this represents removing the digit from the final number). Once it is no longer smaller, or there are no more elements left in the stack, or we have reached the maximum  $k$  deletions, then place  $a_i$  onto the top of the stack. Finally, read off the stack from bottom to top in order to get our final number. If this number is still longer than  $n - k$  digits, this would imply that it is sorted in non-decreasing order, so we can delete everything but the first  $n - k$  digits.

Each digit is added onto the stack once, and can be deleted off the stack at most once. All stack operations are  $O(1)$  time, so the final complexity is  $O(N)$ .

*Proof of Correctness:*

Consider the optimal solution  $B$  equal to  $b_{n-k-1}b_{n-k-2}\dots b_0$ . We show that our algorithm makes the exact effort required to bring a digit equal to  $b_{n-k-1}$  to the front, then  $b_{n-k-2}$  to second place, and so on. Suppose  $a_i$  is the leftmost digit in  $n$  that is equal to  $b_{n-k-1}$ . Then there is always an optimal solution where  $b_{n-k-1}$  corresponds to  $a_i$  in the original number. If there were not, and  $b_{n-k-1}$  corresponded to some other digit  $a_j = a_i$  where  $j > i$ , then we could simply replace  $a_j$  with  $a_i$  to get a solution just as good.

We show that our algorithm will always bring  $a_i$  to the bottom of the stack. Firstly, every element from  $a_{n-1}$  to  $a_{i+1}$  must be larger than  $a_i$ . They cannot be equal by the definition of  $a_i$ , and they cannot be smaller since if there existed an  $a_j < a_i$  where  $j < i$ , then we could improve the optimal solution to be better (i.e.  $a_jb_{n-k-2}\dots b_0$ ), which is a contradiction. Now since everything before  $a_i$  is bigger than  $a_i$ , then right before  $a_i$  is placed on the stack, everything in the stack will be removed by our algorithm. Furthermore, it's guaranteed that we can do so within  $k$  deletions since if we couldn't, then it would be impossible for a digit equal to  $b_{n-k-1}$  to be at the front of the optimal solution. In other words, our algorithm always places  $a_i$  at the bottom of the stack at one point.

Next we show that our algorithm never removes  $a_i$  after it is placed there. For  $a_i$  to be removed, that implies that there is another digit  $a_j$  after  $a_i$  that is the first digit in the final number generated by our algorithm. This implies that every digit before  $a_j$  (including  $a_i$ ) must have been greater than  $a_j$ , since if they weren't, we could take the smallest out all of these numbers, and see that our algorithm would never have deleted it off the stack in order to put  $a_j$  at the bottom. Therefore  $a_j$  is smaller than  $a_i$ . But this

would imply that our algorithm found a solution that is better than the optimal solution, which once again is a contradiction. Hence  $a_i$  is never removed.

But now we have shown that it is always optimal to use the leftmost occurrence of a digit  $a_i$  equal to  $b_{n-k-1}$  to bring to the front. Furthermore, we've shown that our algorithm always does so in  $i - 1$  deletions, and keeps it there. In other words, how we handle the first  $i$  digits in  $A$  is optimal. But now that we know this, we can solve a new subproblem: What is the minimum number we can get from digits  $a_{i-1}a_{i-2}\dots a_0$  after deleting  $k - (i - 1)$  digits? This is a smaller instance of the same problem, and we can simply continue on with our current algorithm to solve it. Thus our greedy algorithm will always find an optimal solution.

## Question 11

You are given an integer  $n$  and a sequence of  $n$  books on a bookshelf. The  $i$ th book has  $p_i$  pages. Define the *disjointedness* of the sequence of books to be

$$D = \sum_{i=1}^{n-1} |p_{i+1} - p_i|.$$

You are also given a positive integer  $k < n$ . You can remove up to  $k$  books from the sequence, while keeping the rest in their original order.

Design an algorithm which finds the minimum disjointedness of the bookshelf with up to  $k$  books removed, and runs in  $O(nk^2)$  time.

The input consists of the positive integers  $n$  and  $k$  where  $k < n$ , as well as  $n$  positive integers  $p_1, \dots, p_n$ .

The output is a single non-negative integer, the minimum disjointedness.

For example, if  $n = 7$  and the sequence of  $p_i$  is  $\langle 6, 8, 4, 2, 7, 3, 9 \rangle$ , the disjointedness is originally 19. If  $k = 2$ , we can remove up to two books. Removing the second and sixth books gives a disjointedness of 11, which is the smallest value that can be achieved.

## Solution

### Subproblems

Let  $P(i, j)$  be the problem of determining  $\text{opt}(i, j)$ , the minimum disjointedness of a sequence of  $j$  books ending at the  $i$ th book.

### Recurrence

For all  $1 < j \leq n$ , we have

$$\text{opt}(i, j) = \min_{j-1 \leq t < i} (\text{opt}(t, j-1) + |p_i - p_t|).$$

We iterate over all choices  $t$  for the index of the penultimate remaining book, and add the difference in pages between book  $t$  and book  $i$ .

### Base cases

The base cases are when  $j = 1$ , i.e. when there is no penultimate book. Clearly  $\text{opt}(i, 1) = 0$  for all  $i$ .

### Overall solution

The solution to  $P(i, j)$  depends on the solutions to various  $P(t, j-1)$ , so we fill the grid from left to right, i.e. we solve all  $P(i, 1)$  then all  $P(i, 2)$  and so on.

The overall solution is the best way to delete up to  $k$  books, i.e. leave at least  $n - k$  remaining books. Thus the answer is

$$\min_{n-k \leq i \leq n, j \geq n-k} \text{opt}(i, j).$$

### Time complexity

Note that in any subproblem  $P(i, j)$ , we must have  $i - j \leq k$ , as otherwise we would have already deleted more than  $k$  books. Thus for each of  $O(n)$  values of  $i$  there are only  $O(k)$  subproblems, and each is solved in  $O(k)$ , giving a total time complexity of  $O(nk^2)$ .

## Question 12

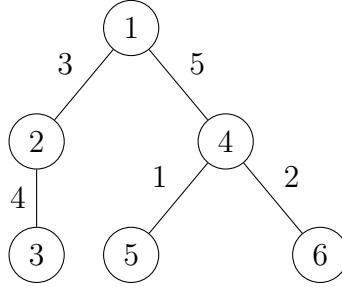
You are given a tree with  $n$  vertices, where vertex 1 is the root. Each edge  $e$  has an associated value, which is the cost to cut this edge.

Design an algorithm which runs in time polynomial in  $n$  and determines a set of edges of minimum total cost such that cutting all chosen edges would disconnect all leaf vertices from the root.

The input consists of the positive integer  $n$ , as well as  $n - 1$  pairs  $(p_2, c_2), \dots, (p_n, c_n)$ , denoting for each vertex  $i$  other than the root its parent  $p_i$  and the cost  $c_i$  of the edge between  $p_i$  and  $i$ .

The output consists of a subset of the edges of this graph. These edges may be presented in any data structure or format.

For example, suppose the tree is as pictured below.



The best set of edges to cut is  $\{(1, 2), (4, 5), (4, 6)\}$  for a total cost of 6.

## Solution

Construct a flow network with:

- source  $s$  and sink  $t$
- a vertex  $v_i$  for each vertex of the tree
- an edge from  $s$  to  $v_1$  with capacity  $\infty$
- for each  $i > 1$ , an edge from  $v_{p_i}$  to  $v_i$  with capacity  $c_i$
- for each leaf vertex  $i$  of the tree, an edge from  $v_i$  to  $t$  with capacity  $\infty$ .

Any finite-capacity cut in this graph will disconnect the source from the sink (since the edges from  $s$  and to  $t$  have infinite capacity). To minimise the cost, we must look for a minimum cut.

To find which edges to cut, we first run the Edmonds-Karp algorithm, and after it terminates we perform one more breadth-first search from  $s$ . The minimum cut will then be  $(S, T)$  where  $S$  contains all vertices seen in this search, and  $T$  contains all others (including  $t$ ). We disconnect those wires which go from a vertex in  $S$  to a vertex in  $T$ .

Constructing the graph takes  $O(V + E)$  time, and finding a minimum cut takes  $O(VE^2)$ . Since  $V = n + 2$  and  $E = 1 + (n - 1) + \#\text{(leaves)} \leq 2n$ , the time complexity is clearly polynomial in  $n$ .