KØBENHAVNS UNIVERSITET

Introduktion til diskret matematik og algoritmer: Exam April 10, 2024
Problems and Solutions

Please note that the main purpose of this document is to explain what the correct solutions are and how to arrive at them. Thus, while the text below is certainly intended to provide good examples of how to solve problems and reason about solutions, these examples do not necessarily specify exactly how the handed-in exams were expected to look like. This is especially so since for many problems there are more than one correct way of solving them. As communicated during the course, the course notes published on Absalon, which contain many problems that we worked out in class, are probably the best indicator of what level of detail is expected from the exam solutions.

1 (70 p) In the following snippet of code `A` is an array indexed from 1 to n containing elements that can be compared and (totally) ordered using the operators $=$, $>$, and $<$.

```
i    := 1
good := TRUE
while (i < n and good)
    j := n
    while (j > i and good)
        if (A[i] != A[j])
            j := j - 1
        else
            good := FALSE
    if (good)
        i := i + 1
if (good)
    return TRUE
else
    return (i, j)
```

1a (30 p) Explain in plain language what the algorithm above does. When and why does the algorithm return *TRUE*? What is the meaning of the pair (i, j) returned instead of *TRUE*?

Solution: The algorithm will let i increase from 1 to n and, for every i, let j decrease from n down to i. For every pair (i, j) the algorithm will check whether the elements $A[i]$ and $A[j]$ stored in the array in positions i and j are distinct. If all checks succeed, i.e., if all elements are distinct, then *TRUE* will be returned. Otherwise, the algorithm will return a pair (i, j) such that $A[i] = A[j]$.

1b (10 p) Provide an asymptotic analysis of the running time as a function of the array size n.

Solution: All lines except for the while loops take a constant amount of time, so in order to determine the time complexity we just need to count how many times the lines in the loops are executed.

The worst-case running time is attained when *good* is always *TRUE*, since this causes the while loops to iterate for the maximal number of steps. When this is the case, the outer while loop runs for i from 1 to n for a total of n steps, and the inner loop runs for $n - i$ steps. The total running time will thus be proportional to $\sum_{i=1}^{n} i = n(n+1)/2$, and so the time complexity of the algorithm is $\mathrm{O}(n^2)$ (or even $\Theta(n^2)$ if you wish to be more precise).

1c (30 p) Can you improve the code to run faster while retaining the same functionality? Note that you can compare elements with operators $=$, $>$, and $<$, but other than that it is not known what the elements are. If you are able to provide a more efficient algorithm, how much faster can you get it to run? Analyse the time complexity of any new algorithm.

Suppose that we are guaranteed that all elements in the array `A` are integers between $-n$ and n. Can you then improve the code to run faster while retaining the same functionality? If you are able to provide a more efficient algorithm, how much faster can you get the algorithm to run? Analyse the time complexity of any new algorithm. Can you prove that it is asymptotically optimal?

Solution: In the general case, we can always sort the elements using, e.g., heap sort in time $\mathrm{O}(n \log n)$. In order to be really careful, we could create an auxiliary array B initialized so that $B[i] = i$. Whenever we swap positions of elements in A, we perform exactly the same swaps in B, so that it is always the case that the original position of the element currently stored in $A[i]$ can be found in $B[i]$.

Once the array A is sorted, we can make a linear scan over it to see if we can find an $i \in [n-1]$ such that $A[i] = A[i+1]$. If so, we return $(B[i], B[i+1])$, and otherwise we return true. The total running time of this algorithm is $\mathrm{O}(n \log n)$. (It is far beyond the scope of this course, but it is possible to prove that if all we can do is to compare elements, then $\Omega(n \log n)$ is also a lower bound, basically for similar reasons as to why this is a lower bound for comparison-based sorting.)

Suppose now that all elements in the array A are guaranteed to be integers between $-n$ and n. Then by using an auxiliary array B, which, for simplicity, we index from $-n$ to n, for every integer $s = A[i]$ in $[-n, n]$ we can store in $B[s]$ the position i in which this integer was found, and terminate as soon as we find a collision. The pseudocode becomes something like the following:

```
for i := -n upto n
    B[i] := -1  // initialize to invalid position
i    := 1
good := TRUE
while (i < n and good)
    if (B[A[i]] == -1)
        B[A[i]] := i
        i := i + 1
    else
        good := FALSE
if (good)
    return TRUE
else
    return (i, B[A[i]])
```

This piece of code has one for loop that will run for $2n + 1$ steps and one while loop that will run for a maximal number of n steps. All operations performed take constant time. The time complexity is therefore $\mathrm{O}(n)$. This is asymptotically optimal, since in order to detect whether there are to copies of the same element in an array of size n we have to at least read all of the input, and this takes time proportional to the size of the input, i.e., $\Omega(n)$.

2 (60 p) Provide formal proofs of the following claims using proof techniques that we have learned during the course.

2a (30 p) For all $K \in \mathbb{Z}^+$ it holds that $\sum_{s=1}^{K} s \cdot s! = (K+1)! - 1$.

Solution: We prove this equality by induction over K.

Base case: For $K = 1$ we have $\sum_{s=1}^{1} s \cdot s! = 1 \cdot 1! = (1+1)! - 1 = 1$.

Induction step: Assume that the equality holds for K and consider $K + 1$. We have

$$\begin{aligned}
\sum_{s=1}^{K+1} s \cdot s! &= \sum_{s=1}^{K} s \cdot s! + (K+1)(K+1)! \\
&= (K+1)! - 1 + (K+1)(K+1)! && \text{[by the induction hypothesis]} \\
&= (K+2)(K+1)! - 1 && \text{[rearranging terms]} \\
&= (K+2)! - 1
\end{aligned}$$

which is the desired equality.

The claim now follows by the induction principle.

2b (30 p) For all $q \in \mathbb{N}$ it holds that $2^{4q+2} + 3^{q+2}$ is a multiple of 13.

Solution: We prove this by induction over q.

Base case: For $q = 1$ we have $2^6 + 3^3 = 64 + 27 = 91 = 13 \cdot 7$.

Induction step: Suppose that $13 \mid 2^{4q+2} + 3^{q+2}$. Equivalently, this means that there is some integer N such that $2^{4q+2} + 3^{q+2} = 13N$. Fixing this N, and working on the expression for $q + 1$, we can then write

$$\begin{aligned}
2^{4(q+1)+2} + 3^{(q+1)+2} &= 16 \cdot 2^{4q+2} + 3 \cdot 3^{q+2} \\
&= 3 \cdot \left(2^{4q+2} + 3^{q+2}\right) + 13 \cdot 2^{4q+2} && \text{[rearranging terms]} \\
&= 3 \cdot 13N + 13 \cdot 2^{4q+2} && \text{[by the induction hypothesis]} \\
&= 13 \cdot \left(3N + 2^{4q+2}\right)
\end{aligned}$$

which shows that this expression is divisible by 13.

The claim follows by the induction principle.

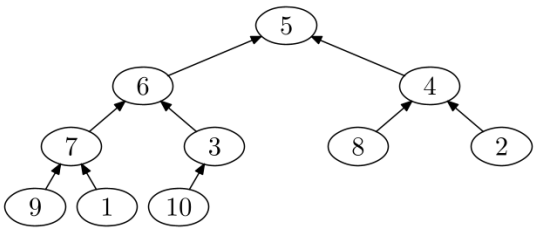
3 (60 p) Let us return to the array $A = [5, 6, 4, 7, 3, 8, 2, 9, 1, 10]$ that was considered in problem set 2. Just as in that problem set, we would like to sort A in increasing order.

3a (50 p) Run heap sort by hand on this array for the first few steps of the algorithm. Show how the original heap is built, and how the first two numbers in the array are sorted into their correct final positions. Make sure to explain what calls to the heap are made from the sorting algorithm and what comparisons are made, and illustrate how the heap and array have changed at the end of every "heapify" or "bubble" call (after all recursive subcalls have been taken care of).

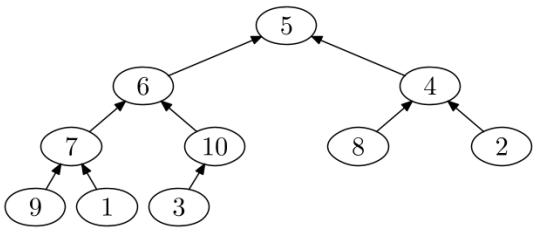
Remark: Please note that you do *not* have to run the whole sorting algorithm, since this might be quite time-consuming. Just showing the initial steps as described above is fully sufficient.

Solution: We illustrate in Figure 1 how the heap used for sorting the array is built and then modified as the first two numbers are extracted.

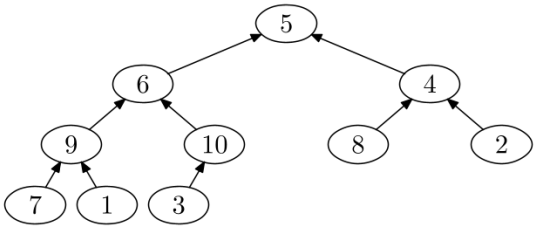
1. The input array is shown in a heap structure in Figure 1a.
2. We need to run BUILD-MAX-HEAP, which builds max-heaps bottom-up using the MAX-HEAPIFY method. The 6th through 10th nodes vacuously constitute max-heaps of size 1, so the first time MAX-HEAPIFY is run is on the 5th node containing 3. Since the child 10 is larger, the elements 3 and 10 swap places as shown in Figure 1b. When MAX-HEAPIFY is called for the subheap rooted at the 10th node, it returns immediately, since that subheap has size 1 and so is a correct max-heap vacuously.
3. MAX-HEAPIFY is then run on the 4th node. The largest child 9 is larger than the number 7 stored in the 4th node, so they trade places as shown in Figure 1c. Again, the second recursive MAX-HEAPIFY call returns immediately.
4. For the 3rd node, the number 4 is smaller than the largest child 8, so the two numbers are swapped and the recursive MAX-HEAPIFY call returns immediately, resulting in the heap in Figure 1d.
5. For the 2nd node, the number 6 is smaller than the largest child 10, so they exchange positions with 6 being shifted to the 5th node. For the recursive MAX-HEAPIFY call on the 5th node, the number 6 is larger than the only child 3 and so the subheap is in order. After this the heap looks as in Figure 1e.
6. In the final MAX-HEAPIFY call of BUILD-MAX-HEAP, the number 5 at the root is compared to both children of the root. The largest child 10 is moved to the root and5 is shifted down to the 2nd node. In the first recursive MAX-HEAPIFY call, 5 is compared to the children 9 and 6, and the largest child 9 is moved up and 5 is shifted down to the 4th node. In the next MAX-HEAPIFY call for the 4th node, 5 is swapped with the largest child 7. The final recursive call for the 8th node terminates since the subheap has size 1. The max-heap resulting from the BUILD-MAX-HEAP call is shown in Figure 1f.
7. The heap sort algorithm now repeatedly removes the root, places the last element in the heap in the root position, and runs MAX-HEAPIFY. Right after 10 has been removed and 3 has been put in the root position, we have the heap structure in Figure 1g. A sequence of recursive MAX-HEAPIFY calls compare 3 first to the largest of the two children 9 and 8 of the root, moving up 9 and moving down 3 to the 2nd node, then to the largest of the



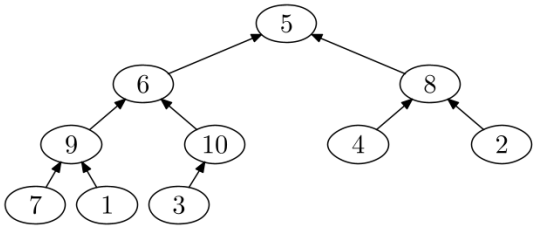
(a) Input array.



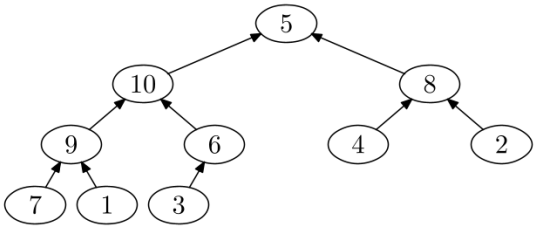
(b) After max-heapify on 5th node.



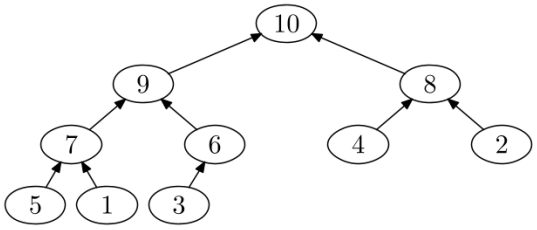
(c) After max-heapify on 4th node.



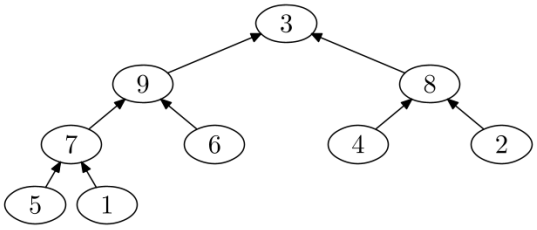
(d) After max-heapify on 3rd node.



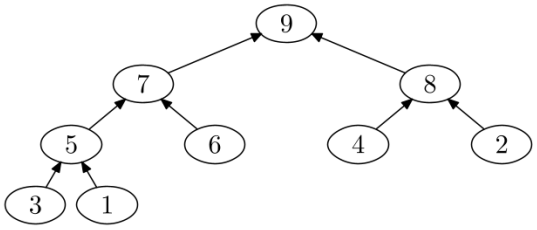
(e) After max-heapify on 2nd node.



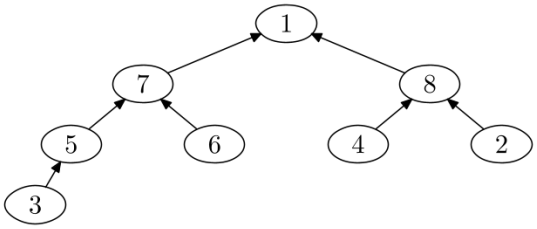
(f) After max-heapify on root node.



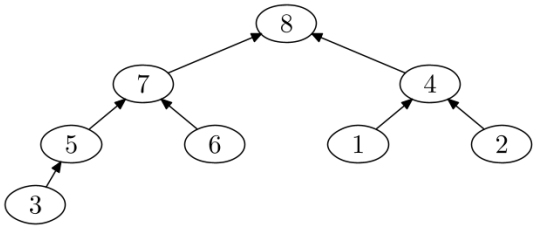
(g) Right after removing largest element 10.



(h) After max-heapify on root again.



(i) Right after removing next-to-largest element 9.



(j) After max-heapify on root again.

Figure 1: Heap configurations for heap sort in Problem 3a.

two children 7 and 6, moving up 7 and moving 3 further down to the 4th node, and finally to the largest of the two children 5 and 1, swapping 3 and 5. This yields the max-heap on nine elements in Figure 1h.

8. Right after 9 has been removed and 1 has been put in the root position, we have the heap structure in Figure 1i. The recursive MAX-HEAPIFY calls compare 1 first to the largest of the two children 7 and 8 of the root, changing places of 1 and 8, and then to the largest of the two children 4 and 2, changing places of 1 and 4. The resulting max-heap on eight elements is depicted in Figure 1h.

This completes the description of how the first two numbers 10 and 9 are extracted from the heap during heap sort, which is what we were asked to explain and illustrate.

3b (10 p) Suppose that we build a max-heap from any given array A. Is it true that once the array A has been converted to a correct max-heap, then considering the elements in reverse order (i.e., $A[n], A[n-1], \ldots, A[2], A[1]$) yields a correct min-heap? Prove this or give a simple counter-example.

Solution: No, this is not true. If we consider the array from our max-heap in Problem 3a, it looks like $10, 9, 8, 7, 6, 4, 2, 5, 1, 3$ (see again Figure 1f). If we take the array in the reverse order to get $3, 1, 5, 2, 4, 6, 7, 8, 9, 10$, then we have that 1 is the left child of the root 3, which violates the min-heap property.

4 (100 p) The busy life as a computer science professor does not leave much time for exercise, but Jakob has finally enlisted the help of a personal trainer (PT) to do something about this. The PT has devised an ambitious program in which Jakob will need to lift dumbbells of weights in all increments of $3, 4, 5, \ldots, 9$ kilos.[1]

The local fitness store sells weights in the range $1, 2, 3, \ldots, 9$ kilos. Jakob would like to buy as few weights as possible so that they can be combined to yield all weight sums $3, 4, 5, \ldots, 9$ kilos in the program suggested by the PT. Just to give a couple of examples, with two 3-kilo weights the possible weight sums are 3 and 6 kilos, while with one 2-kilo weight and one 3-kilo weight the possible weight sums are 2, 3, and 5 kilos. Jakob has given up on trying to get away with buying two weights only, but would really prefer not having to buy more than three weights.

Note: All the subproblems described below can be solved independently of one another! You do *not* need to solve Problem 4a in order to solve Problem 4b, or Problem 4b in order to solve Problem 4c, for instance.

4a (10 p) How many different combinations of three weights in the range $1, 2, 3, \ldots, 9$ are there to consider as Jakob tries to figure out how to buy only three weights that can sum to all desired total weights?

Remark: Please do not just state a number, but also provide an explanation of what this number means and how you know it is the right number. In fact, you do not even have to compute the exact number—answering with a combinatorial expression containing products or quotients of integers and/or factorials of integers, together with an explanation of why this expression is correct, is sufficient.

Solution: Rephrasing the question, we are simply being asked how many different multi-sets of size 3 we can generate from 9 elements. As we learned when we studied combinatorics, the

[1] Well, OK, ambitious or not so ambitious, but you have to start *somewhere*, no?

number of k-multi-sets in a universe of size n is $\binom{n+k-1}{k}$, and so the answer in this case is $\binom{9+3-1}{3} = \frac{11 \cdot 10 \cdot 9}{3 \cdot 2 \cdot 1} = 165$ (but what we really care about is the first of these expressions, so answering $\binom{11}{3}$ together with a clear explanation as above is sufficient).

4b (50 p) Prove that if Jakob wants to buy weights that can be combined to yield all weight sums in the range $3, 4, 5, \ldots, 9$, then unfortunately he will have to buy more than three weights.

Remark: Note that a solution to this problem will have to show conclusively that none of all the combinations in Problem 4a works. Since this is a fairly large number of combinations, the approach will probably have to be something smarter than exhaustive case analysis.

Solution: A first observation is that if there were a solution with only three weights, then all of these weights would have to be different. Note that from 3 weights a, b, and c we can create the $2^3 - 1 = 7$ weight sums

$$a,\ b,\ c,\ a+b,\ a+c,\ b+c,\ a+b+c, \tag{1}$$

and if we are to hit all 7 numbers between 3 and 9 inclusive, then all these weight sums need to be distinct. This in turn means that all numbers a, b, c have to be distinct, because otherwise there would be repetitions in the list (1). Moreover, all numbers have to be in the interval $[3, 9]$, because otherwise one of the sums in (1) will be outside of that range, and the remaining 6 weight sums could not hit 7 distinct numbers.

Consider now any choice of three distinct weights $a < b < c$. It follows from what was just said that $a = 3$. This is so since if $a < 3$, then the first number in (1) is outside of the range $[3, 9]$ and there is no way the remaining 6 weight sums can cover 7 numbers. Also, if $a > 3$, then we are (obviously) failing to generate the weight sum 3.

Since we must also generate the weight sum 4, we must have $b = 4$. Similarly, since $3 + 4 = 7 > 5$ we must set $c = 5$ to hit 5. But now $a + b + c = 3 + 4 + 5 = 12$ is outside of the range $[3, 9]$, so we cannot be hitting all numbers in that range (and indeed, by quick inspection we find that while we can generate $7 = 3 + 4$, $8 = 3 + 5$, and $9 = 4 + 5$, there is no way to get 6).

Since we have proven above that any solution with just three weights would need to use the weights 3, 4, and 5, but since this suggested solution does not in fact work, this shows that it is not possible to generate all weight sums in $[3, 9]$ with just three weights.

4c (20 p) Show that it is sufficient for Jakob to buy four weights in order to be able to combine them to yield all weight sums $3, 4, 5, \ldots, 9$.

Solution: There are many solutions here. Let us list a few:

1. The set of weights $\{1, 2, 3, 4\}$ generates all numbers between 1 and 10 (including $[3, 9]$).
2. The set of weights $\{2, 3, 4, 5\}$ generates all numbers between 2 and 12 plus 14 (including $[3, 9]$).
3. The set of weights $\{3, 4, 5, 6\}$ generates also $7 = 3 + 4$, $8 = 3 + 5$, $9 = 3 + 6$, $10 = 4 + 6$, $11 = 5 + 6$, $12 = 3 + 4 + 5$, $13 = 3 + 4 + 6$, $14 = 3 + 5 + 6$, and $15 = 4 + 5 + 6$, i.e., all weights in the range $[3, 15]$, plus in addition $18 = 3 + 4 + 5 + 6$.

Listing just one solution here is enough for a full score on this subproblem (although Jakob is of course grateful for the information that variant 3 seems like a particularly inspired choice if his new training regimen would lead to unexpected progress).

4d (20 p) Jakob also wants to minimize the *total weight* of all the weights he buys, i.e., the sum of the weights, in order not to have to carry around too heavy a training bag. What is the smallest possible total weight of a collection of four weights that can be combined to yield all weight sums $3, 4, 5, \ldots, 9$ as in Problem 4c?

Solution: Looking at variant 1 above, with the set of weights $\{1, 2, 3, 4\}$, but trying to minimize the total weight, we can try the multi-set $[1, 2, 3, 3]$. It is straightforward to verify that from this set we can generate all weight sums in the range $[3, 9]$. Note also that the total sum of these weights is 9 kilos, which is clearly necessary if we are to be able to generate the weight sum 9. Hence the answer is that a total weight of 9 kilos is both necessary and sufficient.

5 (60 p) During the stressful mornings when Jakob is teaching the IDMA course in Copenhagen, he needs to successfully plan the following actions:

Coffee Have a morning latte.

Dress Put on clothes.

Lecture Give the IDMA lecture in Copenhagen.

Shower Take a shower.

Train Take the train from Lund to Copenhagen.

Wake up The hardest task of them all...

Just a quick look at this list reveals that the correct sequencing of actions is crucial in order to avoid potentially catastrophic consequences. Jakob is aware of the following constraints:

- He has to take the **train** to Copenhagen before **lecturing**.
- He also needs to have **coffee** before **lecturing**.
- He has to **dress** before taking the **train**.
- He has to **shower** before **dressing**.
- He has to **wake up** before taking a **shower**.
- He definitely has to **dress** before **lecturing**.
- Very annoyingly, he has to **wake up** before having **coffee** (though the other way round would be more helpful).
- Finally, ideally he should also **wake up** before **lecturing**.

5a (10 p) Model this problem as a graph, with the constraints corresponding to edges between actions. Make sure to explain concretely what graph you get for Jakob's problem.

Solution: Following the instructions, we create a graph with six vertices, corresponding to the six actions, with each vertex labelled by the initial letter of the action. Every constraint that action A has to happen before action B corresponds to a directed edge from A to B (so that, for instance, having to take the Train before Lecturing corresponds to a directed edge from T to L, and having Coffee before Lecturing corresponds to a directed edge from C to L). This yields the graph in Figure 2.

5b (50 p) Propose a suitable graph algorithm to solve Jakob's problem. Explain what this algorithm is and why it is the right choice for this problem. Make a dry-run of the algorithm and explain the relevant steps in the execution (similarly to what has been done in class and in the lecture notes). Let your graph process all actions in alphabetical order as listed above. What morning schedule can you propose for Jakob based on what your algorithm says?

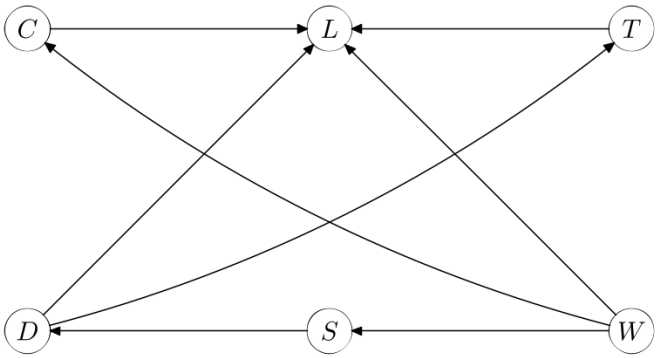


Figure 2: Directed graph modelling scheduling problem in Problem 5.

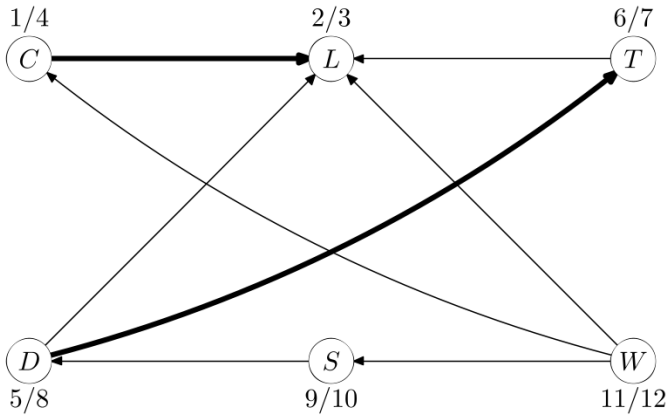


Figure 3: Result of depth-first search with DFS forest and start and finishing times.

Solution: What Jakob is looking for is a scheduling of the actions that respects all dependencies. In graph theory terms, this is the same as finding a topological sort of the directed graph in Figure 2, if such a topological sort exists.

We have learned that we can use depth-first search to produce a topological sort of the graph (or detect that the vertices cannot be sorted topologically, namely if during the depth-first search we discover a back edge to a vertex that has been discovered but not yet finished, because that yields a cycle). Following the instructions in the problem statement, we run the topological sort algorithm on the graph, processing the vertices in alphabetical order (and counting time from 1 upwards, as done in CLRS):

1. At time 1 we discover vertex C, which has the only out-neighbour L.
2. Since vertex L is not yet processed, we discover it from vertex C at time 2, but since the vertex has no out-neighbours we immediately finish processing at time 3, returning to C.
3. Since vertex C has no more out-neighbours, we finish processing of C at time 4.
4. Next in alphabetical order is vertex D, which has not been processed, so we disover D at time 5. The out-neighbour L has already been processed, but not the out-neighbour T.
5. We discover vertex T from vertex D at time 6. The only out-neighbour L of T has already been processed, so T is immediately finished at time 7 and we return to D.

6. Since D has no more out-neighbours, we finish the processing of this vertex at time 8.

7. Next in alphabetical order is L, which has already been processed. After that comes vertex S, which we discover at time 9. The only out-neighbour of S is D, which is already processed, so we immediately finish S at time 10.

8. The next vertex in alphabetical order is T, which has already been processed. The final vertex W is discovered at time 11, and since all other vertices in the graph have been processed at this point we finish processing immediately at time 12.

The start and finishing times, and the resulting depth-first search forest, are presented in Figure 3. Since we discovered no back edges, reading off the vertices in decreasing order of finishing times yields the topological sort *Wake up – Shower – Dress – Train – Coffee – Lecture*.

6 (60 p) In this problem we focus on relations. In particular, suppose that $A = \{e_1, e_2, \ldots, e_6\}$ is a set of 6 elements and consider the relation R on A represented by the matrix

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

(where element e_i corresponds to row and column i).

6a (20 p) Let us write T to denote the transitive closure of the relation R. What is the matrix representation of T in this case? Can you explain clearly in words what the relation T is?

Solution: In R, we have that e_i is related to e_{i-1}. By transitivity, this gives us that e_i is related to e_{i-2}, and by induction we see that e_i is related to e_j for $i > j$, yielding the matrix

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

which is the matrix for a linear order relation (such as greater-than).

6b (20 p) Suppose that we create a new relation S_1 from R by first taking the reflexive closure and then the symmetric closure. What does the matrix representation of S_1 look like? Can you explain clearly in words what the relation S_1 is?

Solution: Taking the reflexive closure means adding an all-1s diagonal to the matrix for the relation (regardless of what the matrix is). Since e_i is related to e_{i-1} in R, taking the symmetric closure means that we will get that e_{i-1} is related to e_i also, while the all-1s diagonal is not affected. This leads to the matrix

$$M_{S_1} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

which shows that e_i is related to e_j if and only if $|i - j| \leq 1$.

6c (20 p) Suppose instead that we first take the symmetric closure and then the reflexive closure to derive another relation S_2 from R. Are S_1 and S_2 different relations, or are they the same relation in the end? If the latter case holds, is it true for any relation R on a set A that relations S_1 and S_2 derived in this way will be the same, or can they sometimes be different? Give a proof or present a simple counter-example.

Solution: As already written in the solution for Problem 6b, the reflexive closure just adds an all-1s diagonal, regardless of when this closure operator is applied, and does not affect any off-diagonal elements in the matrix representation. The symmetric closure adds the pair (e_j, e_i) for any pair (e_i, e_j) already in the relation, and thus leaves all diagonal entries unchanged. Therefore, the order in which symmetric and reflexive closures are taken does not matter, and so it is true for any relation R that the relations S_1 and S_2 constructed as described above will be the same.

7 (110 p) In this problem we wish to study graphs viewed as relations.

7a (10 p) Recall that two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are said to be *isomorphic*, denoted $G \cong H$, if there is a bijection $f : V_G \to V_H$ such that $(u, v) \in E_G$ holds if and only if $(f(u), f(v)) \in E_H$.

Argue, briefly but clearly, why $\cong$ defines an equivalence relation.

Solution: An equivalence relation is a relation that is reflexive, symmetric, and transitive. Pointing out these conditions clearly, and explaining briefly why they hold this this case, is fully sufficient, but let us belabour the point a little bit below just for clarity.

Graph isomorphism $\cong$ is clearly reflexive, since the identity map i sending any $v \in V_G$ to v itself is a bijection vacuously satisfying the condition that $(u, v) \in E_G$ if and only if $(i(u), i(v)) = (u, v) \in E_G$.

The relation is also symmetric, since if f is a bijection $f : V_G \to V_H$, then it is easy to see that the inverse function $f^{-1} : V_H \to V_G$ is a bijection that exactly preserves all edge relations.

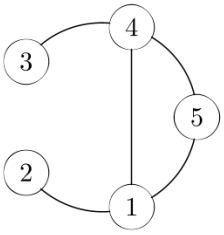
Finally, if $G \cong G'$ and $G' \cong G''$ as witnessed by bijections $f : V_G \to V_{G'}$ and $f' : V_{G'} \to V_{G''}$, respectively, then it is straightforward to verify that the composition $f' \circ f$ is a bijection between $V(G)$ and $V(G'')$ that exactly preserves edges, witnessing that $G \cong G''$ holds.

7b (50 p) Consider the graphs in Figure 4. Explain how the relation $\cong$ partitions these graphs into equivalence classes. Argue, briefly but clearly, why any pair of graphs G_i and G_j for $1 \leq i < j \leq 4$ are or are not in the same equivalence class, respectively.

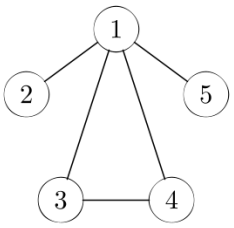
Solution: Any equivalence relation R on a (finite) set $\mathcal{A}$ partitions $\mathcal{A}$ into a (finite) collection of subsets $A_i \subseteq \mathcal{A}$ such that $A_i \cap A_j = \emptyset$ for $i \neq j$ and $\mathcal{A} = \bigcup_i A_i$, where for every A_i it holds that $a, b \in A_i$ if and only if $(a, b) \in R$.

What this means concretely in this problem is that we are asked to partition the graphs in Figure 4 into subsets of isomorphic graphs. If two graphs are isomorphic, then we need to exhibit an isomorphism to show this. If two graps are *not* isomorphic, then we need a proof why there can exist no isomorphism between them. Let us try to analyse the different graphs.

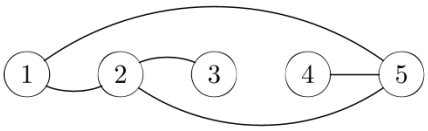
- The graph G_2 cannot be isomorphic to any of the other graphs, and so is in its own subset in the partition. One way to see this is that vertex 1 has 4 neighbours, and any graph isomorphism would need to map this vertex to another vertex of degree 4. However, none of the other graphs has any vertex of degree larger than 3.



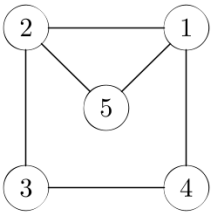
(a) Graph G_1.



(b) Graph G_2.



(c) Graph G_3.



(d) Graph G_4.

Figure 4: Graphs for Problem 7.

- The graph G_4 cannot be isomorphic to any of the other graphs either, and so is in its own subset in the partition. One reason for this is that all vertices in G_4 have degree at least 2, but all other graphs have some vertices of degree 1. Another possible argument is that G_4 contains a (simple) cycle of length 4, which would need to map to another (simple) cycle of length 4 under any isomorphism, but none of the other graphs has a simple cycle longer than 3. Yet another argument is that G_4 has strictly more edges than all the other graphs.

- It only remains to figure out whether $G_1 = (V_1, E_1)$ and $G_3 = (V_3, E_3)$ are isomorphic or not. Any edge-preserving bijection $f : V_1 \to V_3$ would have to send degree-1 vertices to degree-1 vertices, so, for instance, it must hold that $f(3) \in \{3, 4\}$. Let us try with $f(3) = 3$. Then for the second degree-1 vertex 2 in G_1 it must hold that $f(2) = 4$. Since isomorphisms have to map neighbours to neighbours, and since neighbours of degree-1 vertices are unique, it follows that we must set $f(4) = 2$ and $f(1) = 5$. But this leaves only one vertex on each side, so we are forced to also set $f(5) = 1$.

  We need to check whether f as defined in this way is a graph isomorphism. It is sufficient to check that all edges in G_1 map to edges in G_3, since the two graphs have the same number of edges (and could not be isomorphic otherwise). Looking at the edges one by one, we have the following conclusions:

  - $(3, 4) \in E_1$ maps to $(f(3), f(4)) = (3, 2) \in E_3$;
  - $(4, 5) \in E_1$ maps to $(f(4), f(5)) = (2, 1) \in E_3$;
  - $(1, 4) \in E_1$ maps to $(f(1), f(4)) = (5, 2) \in E_3$;
  - $(1, 5) \in E_1$ maps to $(f(1), f(5)) = (5, 1) \in E_3$;
  - $(1, 2) \in E_1$ maps to $(f(1), f(2)) = (5, 4) \in E_3$.

  This shows that f is indeed an isomorphism between G_1 and G_3.

We can see from this that we get the partition $\{\{G_1, G_3\}, \{G_2\}, \{G_4\}\}$ of the graphs in Figure 4.

7c (50 p) Let A_G denote the adjacency matrix of a graph G on n vertices. Recall that for any graph G we define $G^\top$ to be the graph with adjacency matrix $A_G^\top$. Define G^i as the graph with adjacency matrix $(A_G)_\odot^i = A_G \odot A_G \odot \cdots \odot A_G$ (with the matrix A_G repeated i times). Finally, let K_n denote the complete graph on n vertices with all possible edges present, corresponding to the adjacency matrix with all entries equal to 1 except for 0s on the diagonal.

When playing around with these notions, Jakob has come up with the following claims. Explain which of them are true or false and why.

1. The edge relation on $V = V(G)$, saying that $E(u, v)$ holds precisely when $(u, v) \in E$, is a transitive relation if and only if G^2 is a subgraph of G.
2. If the edge relation is symmetric, then $G \cong G^\top$.
3. The graph G is connected if and only if $G^n \cong K_n$.
4. The edge relation is asymmetric if and only if G is a directed acyclic graph (DAG).

Solution: Let us consider the statements one by one:

1. This statement is **true**, as can be determined by reading pages 145–146 in KBR, but let us write out the details.

   The edge relation is transitive precisely if whenever (u, v) and (v, w) are edges in G, it also holds that (u, w) is an edge in G (this is how transitivity is defined). We have that (u, w) is an edge in G^2 precisely when there exists some v such that (u, v) and (v, w) are edges in G (again by definition, this time of Boolean matrix multiplication). If the edge relation $E(G)$ defining G is transitive, this means that (u, w) has to be an edge in G as well, meaning that G^2 is a subgraph of G. In the other direction, if G^2 is a subgraph of G, then for any $(u, v), (v, w) \in E(G)$ it holds that $(u, w) \in E\big(G^2\big) \subseteq E(G)$, showing that the edge relation is transitive.
2. This statement is also **true**. If the edge relation is symmetric, then G and $G^\top$ are in fact the same graph, and so definitely isomorphic to each other.
3. This statement is **false**. There is an edge $(u, w) \in E\big(G^n\big)$ precisely when there is a walk (i.e., a not necessarily simple path) of length *exactly* n between u and v. But just because a graph is connected, this does not mean that it is possible to walk exactly n steps to get between any two vertices in the graph.

   Another, perhaps slightly less enlightening, reason why the statement is false is that, in general, G^n may contain self-loops, namely if there are a (possibly non-simple) cycles of length exactly n, whereas K_n does not contain self-loops by definition.
4. This statement is **false**. Consider a directed graph on n vertices that just consists of the cycle $1 \to 2 \to 3 \to \cdots \to n - 1 \to n \to 1$. The edge relation for this graph is asymmetric, but the graph is very clearly not acyclic.

8 (100 p) During his years of teaching introductory discrete mathematics courses, Jakob has developed a not-always-completely-healthy interest in card games. His latest obsession is the game *Stop on Black*, which is played between a dealer and a gambler as described below.

The dealer shuffles a complete deck of 52 cards perfectly, and then deals cards face up from the deck, one card at a time. Before any card is dealt, including at the very start of the game, the gambler can place a bet. Such a bet is placed only once, and has to be made at the very latest before the last card is dealt (or, phrased differently, if the bet has not been placed before,

then it is automatically placed on the last card). After the bet has been placed, the dealer deals the next card, i.e., the one on which the bet has been placed, and the game ends. The gambler wins the game if this card is black (spades or clubs) and loses if it is red (hearts or diamonds).

Jakob is interested in determining what is the highest probability of winning that can be achieved with any gambler strategy (where, to simplify matters, we say that the gambler cannot flip coins or use any other kind of randomness, but has to choose actions only based on the cards seen so far). Jakob has played this game obsessively, but has never been able to win more than half of the time on average (which is trivial to achieve by just placing the bet immediately, before even the first card is dealt).

Jakob claims to have made an exhaustive case analysis on a small deck with 2 red and 2 black cards, and says that (a) this case analysis proves rigorously that the best winning probability is exactly 50% and that (b) this should generalize to any deck of N red and N black cards. Unfortunately, his proof for (a) is pretty much unreadable, and the claim in (b) that it should generalize is even harder to understand.

Can you help Jakob by figuring out whether he is right in claiming that a 50% winning probability is best possible for the gambler, or could you provide a counter-example?

Remark: For a full score, analysing the case of a standard deck with 26 red and 26 black cards is sufficient. Partial results, including proofs or counter-examples for (much) smaller decks of cards (such as what Jakob claims to have analysed), can also yield points.

Solution: People who have followed Jakob's discrete math and algorithm adventures over the years know that his "insights" can be a mixed bag indeed, but this particular case happens to be an example of when he is actually correct.

It is fairly straightforward to show by induction that for any deck with B black and R red cards the probability of winning is always $\frac{B}{B+R}$. Proving this is sufficient for a full score, and the solution can be quite brief.

Before presenting this brief solution, however, let us see how, if one does not find this brief, general solution, one can still provide an analysis of some special cases (as suggested in the remark) to gain some points, and perhaps also enough insights to formulate a hypothesis for the general case.

Deck with one black and one red card: Regardless of how the gambler plays, the winning probability is exactly $\frac{1}{2}$, which can be seen by the following case analysis:

1. *Bet before first card:* The first card is black with probability exactly $\frac{1}{2}$, which is the probability of winning for this strategy.
2. *Bet after first card:* Note that this means, by definition, that the bet has to be placed on the second card. If the first card is black, which happens with probability exactly $\frac{1}{2}$, then the gambler loses with probability 1 since the second card is guaranteed to be red. But if the first card is red, which also happens with probability exactly $\frac{1}{2}$, then the gambler wins with probability 1 since the second card now has to be black. So the probability of winning is exactly $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$.

It follows that *regardless of how the gambler plays*, the winning probability is exactly $\frac{1}{2}$ (no more, but also no less).

Deck with two black and two red cards: Let us do a similar case analysis, where we can use that we have already analysed the game with one black and one red card. We do a case analysis based on whether the bet is placed before or after the first card:

1. *Bet before first card:* The first card is black with probability exactly $\frac{1}{2}$, so that is the exact winning probability for this strategy.
2. *Bet after first card:* If the first card was black, which happens with probability $\frac{1}{2}$, we have the following cases:
   (a) *Bet directly:* The winning probability is $\frac{1}{3}$.
   (b) *Wait at least one more card:* The next card is red with probability $\frac{2}{3}$. But if this happens, we are in the scenario of a one-card-each game, and we know that any gambler strategy in this game has probability exactly $\frac{1}{2}$ of winning. If the next card is black, then the probability of winning is clearly 0. Hence, for any strategy waiting at least one more card after a first black card, the gambler wins with probability $\frac{2}{3} \cdot \frac{1}{2} + \frac{1}{3} \cdot 0 = \frac{1}{3}$.

   In all cases, the winning probability is exactly $\frac{1}{3}$ if the first card was black.

   If instead the first card was red, which also happens with probability $\frac{1}{2}$, we have the following cases:
   (a) *Bet directly:* The winning probability is now $\frac{2}{3}$.
   (b) *Wait at least one more card:* The next card is black with probability $\frac{2}{3}$, leading to a one-card-each game with probability $\frac{1}{2}$ for the gambler of winning. If the next card is also red, then the probability of winning is clearly 1. Hence, for any strategy waiting at least one more card after a first red card, the gambler wins with probability $\frac{2}{3} \cdot \frac{1}{2} + \frac{1}{3} \cdot 1 = \frac{2}{3}$.

   In all cases, the winning probability is exactly $\frac{2}{3}$ if the first card was red.

   This means that since the first card is black or red with 50–50 probability, the winning probability for any strategy that waits at least one card before betting is exactly $\frac{1}{2} \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{2}{3} = \frac{1}{2}$.

It is of course impossible to know whether this is the case analysis Jakob had in mind, but the above reasoning indeed shows for the two-cards-each game that the winning probability of the gambler is exactly $\frac{1}{2}$ (i.e., it is not possible to play better to reach a higher winning probability, but, interestingly, it is also not possible to play worse to reach a lower winning probability).

We are now ready for the general case. We repeat again that a solution for the general case is fully sufficient, and that there is absolutely no need to do a "brute-force analysis" of the two-cards-each game as above if you have found a general solution.

Claim: For a deck with B black and R red cards (for $B + R > 0$), the gambler wins the Stop on Black game with probability $\frac{B}{B+R}$.

This claim is what we want to establish, and we do so by presenting a proof by induction over the total number of cards $B+R$ in the deck, where the claim will serve as our induction hypothesis.

Base case: The claim is clearly true if either $B = 0$ or $R = 0$.

Induction step: Assume that the claim is true for any deck with strictly less than $B+R$ cards. Consider a deck with B black and R red cards, where B and R are both strictly positive integers. We have two cases (mirroring our "brute-force analysis" above):

1. *Bet before first card:* Clearly, the winning probability is exactly $\frac{B}{B+R}$ in this case, since the deck of cards is perfectly shuffled.

2. *Bet after first card:* The first card is black with probability $\frac{B}{B+R}$, in which case we now have a perfectly shuffled deck with $B-1$ black and R red cards. By our induction hypothesis, the gambler wins the game for this deck with probability exactly $\frac{B-1}{B+R-1}$. With probability $\frac{R}{B+R}$, the first card is instead red, leading to a game with B black and $R-1$ red cards. This game is won by the gambler with probability exactly $\frac{B}{B+R-1}$ according to our induction hypothesis.

   It follows that the winning probability for any strategy placing the bet after the first card has been shown is exactly

$$\frac{B}{B+R}\cdot\frac{B-1}{B+R-1}+\frac{R}{B+R}\cdot\frac{B}{B+R-1}=\frac{B(B-1)+RB}{(B+R)(B+R-1)} \tag{2a}$$

$$=\frac{B(B+R-1)}{(B+R)(B+R-1)} \tag{2b}$$

$$=\frac{B}{B+R}\,, \tag{2c}$$

   and so the gambler wins with probability exactly $\frac{B}{B+R}$ in this case as well.

It follows by the principle of mathematical induction that the gambler wins the Stop on Black game with a deck consisting of B black and R red cards with probability exactly $\frac{B}{B+R}$. In particular, for any deck with an equal number of black and red cards the winning probability is exactly 50%, and this is regardless of how intelligent or stupid the gambler strategy is (which might be an explanation of how Jakob managed to consistently reach this winning probability). Note that for a full score it is *not* necessary to argue that the winning probability is exactly 50%, only that this is an upper bound.