KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Exam April 1, 2020
Problems and (Sketches of) Solutions

1 (60 p) In the following snippet of code `A` is an array indexed from 1 to n containing elements that can be compared using the operator $<$.

```
stop := FALSE
i := 1
while (i <= n and not(stop))
    j := 1
    found := FALSE
    fail := FALSE
    while (j <= n and not(fail))
        if (A[i] < A[j])
            if (found)
                fail := TRUE
            else
                found := TRUE
        j := j+1
    if (found and not(fail))
        stop := TRUE
    else
        i := i+1
if (stop)
    return A[i]
else
    return "failed"
```

1a (30 p) Explain in plain language what the algorithm above does.

Solution: The algorithm tries to find the element indexed with i such that there exists exactly one element that has higher value than $A[i]$. If such an element exists, the algorithm returns this element, and otherwise it returns `failed`.

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

Solution: The algorithm has two nested `while` loops. For each element of the array `A` (first while loop) the algorithm compares it with every other element of the array (including itself) to see if certain conditions are satisfied (second while loop). The number of comparisons that is done in the worst case is equal to $n \cdot n$ (this happens, for instance, for an array of distinct elements sorted in increasing order, except that the order of the last two elements is flipped), and the amount of additional work per comparison is easily seen to be at most a constant. Thus, the running time of the algorithm is $\mathrm{O}\big(n^2\big)$ (or, if we want to be more precise, $\Theta\big(n^2\big)$, but just answering with big-oh is perfectly fine).

```
if(A[0]<A[1])
    max = A[1]
    max_2 = A[0]
else
    max = A[0]
    max_2 = A[1]
for(i = 2; i<n; i++)
    if(A[i]>max)
        max_2 = max
        max = A[i]
    else
        if(A[i]>max_2)
            max_2 = A[i]
if(max != max_2)
    return max_2
else
    return "failed"

return 0;
```

Figure 1: Pseudo-code for more efficient version of algorithm in Problem 1.

1c (20 p) Improve the code to run faster while retaining the same functionality. How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Solution: The snippet of code in Figure 1 runs faster while retaining the same functionality. It goes through the array only once, thus having a running time of $O(n)$. Since we want to find the second largest element of the array, we need to go through the array at least one time, meaning that this solution is asymptotically optimal. (This is a general claim that holds in most settings, at least in introductory algorithms classes: In order to be correct, the algorithm has to be given a chance to read the data, and so any linear-time algorithm is asymptotically optimal.)

In the improved version of the algorithm, we instantiate the largest (`max`) and the second largest element (`max_2`) from the first two elements of the array. In the `for` loop we compare the current element of the array with the `max` and `max_2` changing the values of the `max` and `max_2` if necessary. Finally, if `max` and `max_2` are different values we print the `max_2`.

A slightly suboptimal solution would be to instead sort the whole array in time $\mathrm{O}(n \log n)$ and then output the second largest element if if is distinct from the largest element, but this is not asymptotically optimal.

2 (70+ p) In the following snippet of code `A` is an array indexed from 1 to n containing integers.

```
found := FALSE
i := 1
```

```
while (i <= n and not(found))
    j := 1
    while (j <= n and not(found))
        k := 1
        while (k <= n and not(found))
            if (i != j and j != k and i != k and A[i]+A[j]+A[k] == 0)
                found := TRUE
            else
                k := k+1
        if (not(found))
            j := j+1
    if (not(found))
        i := i+1
if (found)
    return (i, j, k)
else
    return "failed"
```

2a (20 p) Explain in plain language what the algorithm above does.

Solution: The algorithm tries to find three distinct indices i, j, k such that $A[i]+A[j]+A[k] = 0$. If such i, j, k exist, the triple (i, j, k) is returned as result, and otherwise the algorithm returns `"failed"`.

2b (10 p) Provide an asymptotic analysis of the running time as a function of the array size.

Solution: The analysis is similar to the one in the previous exercise. We have three nested while loops each iterating from 1 to n. The total amount of work is at most a constant times the number of iterations in the innermost loop. In the worst case (for instance, when there is no solution), our algorithm will make $n \cdot n \cdot n$ iterations of the innermost loop, thus giving us a running time of $\mathrm{O}\big(n^3\big)$.

2c (40 p) Improve the code to run faster while retaining the same functionality. How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Solution: A first, small, optimization would be to let j range from $i+1$ to n and k range from $j+1$ to n in the loops. This does not save asymptotically, but would be a signficant constant factor gain in practice.

We can further improve our algorithm to run in $\mathrm{O}\big(n^2\big)$ time. First we sort the array, which can be done in time $O(n \log n)$. Then we iterate through the array with i ranging from 1 to n. We fix element with index i set $j = i + 1$ and $k = n - 1$ and sum elements $A[i] + A[j] + A[k]$. If the sum is equal to zero, we terminate, otherwise we move either index j or index k based on the result (if the sum is grater than zero we set $k = k - 1$, otherwise we set $j = j + 1$). For each i we go through the whole array, thus our running time is $O(n^2)$. The total running time then is $O(n \log n) + O(n^2) = O(n^2)$.

2d *Bonus problem (worth 40 p extra):* This is in fact a well-known research problem. What information can you find about this on the internet? What are the best known upper and lower bounds for algorithms solving this problem? Explain how you dug up the information, and give references to where it can be found.

Solution: There is no obvious "correct answer" to this literature search problem, but searching on the internet it is not too hard to discover that this is actually an extremely well-studied problem known as the 3-SUM problem. According to Wikipedia, the current best known running time is $\mathrm{O}\big(n^2(\log\log n)^{O(1)}/\log^2 n\big)$. It is believed that it is impossible to solve this problem in time $\mathrm{O}(n^{2-\epsilon})$ for $\epsilon > 0$ in the worst case. Providing this information is sufficient for a full score on this subproblem.

3 (40 p) For positive integers $a_1, a_2, \ldots, a_k$, define $\gcd(a_1, a_2, \ldots, a_k)$ to be the largest positive integer d such that d divides every a_i and any positive integer c that divides every a_i also has to divide d. Is it true that there are integers m_i, not necessarily positive, such that $d = \sum_{i=1}^{k} m_i a_i$? Prove this or give a simple counter-example.

Solution: This is true, and we can prove it by induction on the number of arguments of the function gcd.

Before starting the inductive argument, however, let us first make the observation that $\gcd(a_1, \ldots, a_k, a_{k+1}) = \gcd(\gcd(a_1, \ldots, a_k), a_{k+1})$. To see this, set $d = \gcd(a_1, \ldots, a_k, a_{k+1})$, $A = \gcd(a_1, \ldots, a_k)$, and $d' = \gcd(A, a_{k+1}) = \gcd(\gcd(a_1, \ldots, a_k), a_{k+1})$. We will prove that $d \mid d'$ and $d' \mid d$, meaning that $d = d'$ must hold. By definition, we have $d \mid a_i$ for all i, meaning (trivially) that d must divide both A and a_{k+1}. But, again by definition, any number with that property must divide $\gcd(A, a_{k+1}) = d'$. The other direction is very similar: since $d' \mid A$ we have $d' \mid a_i$ for $i = 1, \ldots, k$, and by assumption $d' \mid a_{k+1}$ also holds. That is, d' divides all the numbers $a_1, \ldots, a_k, a_{k+1}$, but by definition any such number also divides $d = \gcd(a_1, \ldots, a_k, a_{k+1})$ for some integers c_1 and c_2.

We now turn to the inductive proof. The base case is for two arguments, for which we know from the KBR textbook that $\gcd(a_1, a_2)$ can be written as linear combination $\gcd(a_1, a_2) = c_1 a_1 + c_2 a_2$ for some integers $c_1, \ldots, c_k$.

For the induction step, let us assume that for k numbers we can write $A = \gcd(a_1, a_2, \ldots, a_k) = c_1 a_1 + c_2 a_2 + \ldots c_k a_k$. Then for $k+1$ arguments we get

$$
\begin{aligned}
&\gcd(a_1, a_2, \ldots, a_k, a_{k+1}) = \\
&\qquad = \gcd(A, a_{k+1}) && \text{[by the reasoning above]} \\
&\qquad = mA + c_{k+1} a_{k+1} && \text{[by the base case]} \\
&\qquad = m \cdot \gcd(a_1, a_2, \ldots, a_k) + c_{k+1} \cdot a_{k+1} && \text{[just writing it out]} \\
&\qquad = m(c_1 \cdot a_1 + c_2 \cdot a_2 + \ldots c_k \cdot a_k) + c_{k+1} \cdot a_{k+1} && \text{[by the inductive hypothesis]} \\
&\qquad = mc_1 \cdot a_1 + mc_2 \cdot a_2 + \ldots mc_k \cdot a_k + c_{k+1} \cdot a_{k+1}
\end{aligned}
$$

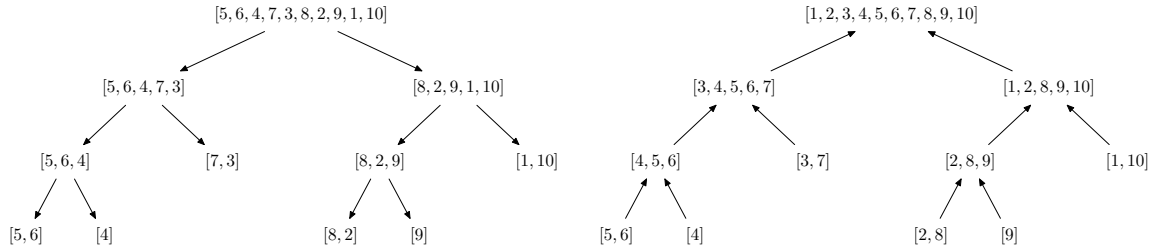
which is an expression exactly on the form we wanted. The claim follows by the induction principle.

4 (110 p) Suppose that we are given an array $A = [5, 6, 4, 7, 3, 8, 2, 9, 1, 10]$ to be sorted in increasing order.

4a (30 p) Run merge sort by hand on this array, and show in detail in every step of the algorithm what recursive calls are made and how the results from these recursive calls are combined.

Solution: See Figure 2 for an illustration of what happens in the algorithm.

We first recursively split the list in two part, where the first part is one element larger if the list size is odd, until all lists have sizes at most 2. This leads to the split lists in the leaves of



(a) Tree of recursive calls splitting list.

(b) Bottom-up sorting of split lists.

Figure 2: Merge sort on array $A = [5, 6, 4, 7, 3, 8, 2, 9, 1, 10]$.

the recursion tree in Figure 2a. These lists are either single elements, which are already sorted, or pairs of elements, that can be sorted by swapping places if needed. After doing this, we have the sorted lists in the leaves of Figure 2b.

In the merge phase we work bottom up from the level above the leaves. Every parent node P takes its two children lists C_1 and C_2, which have previously been sorted, and merges them. We do so by placing to pointers e_1 and e_2 at the start of C_1 and C_2, respectively, and then adding the smallest of these elements to the list under construction after which the corresponding pointer is advanced.

For instance, for the two leftmost leaves we have $e_2 = 4 < 5 = e_1$, so 4 is added first, and since this empties list C_2 we then add 5 and 6 to get the list $[4, 5, 6]$. When $[4, 5, 6]$ is merged with $[3, 7]$, then 3 is the smallest element and goes first. After this, 4, 5, and 6 are all smaller than 7, which goes last. We get the list $[3, 4, 5, 6, 7]$. The recursive calls in the right subtree of the root are dealt with similarly (and this should be described in the solutions).

In the final step, we need to merge $[3, 4, 5, 6, 7]$ and $[1, 2, 8, 9, 10]$. Here we start with $e_2 = 1 < 3 = e_1$, so 1 goes first, updating e_2 to 2. Now we still have $e_2 = 2 < 3 = e_1$, so we add 2 to the list under construction, updating e_2 to 8. Since 8 is larger than all elements in C_2, this whole list will be added, after which 8, 9, and 10 will be added from list C_1. This produces the sorted list at the root of the tree in Figure 2b.

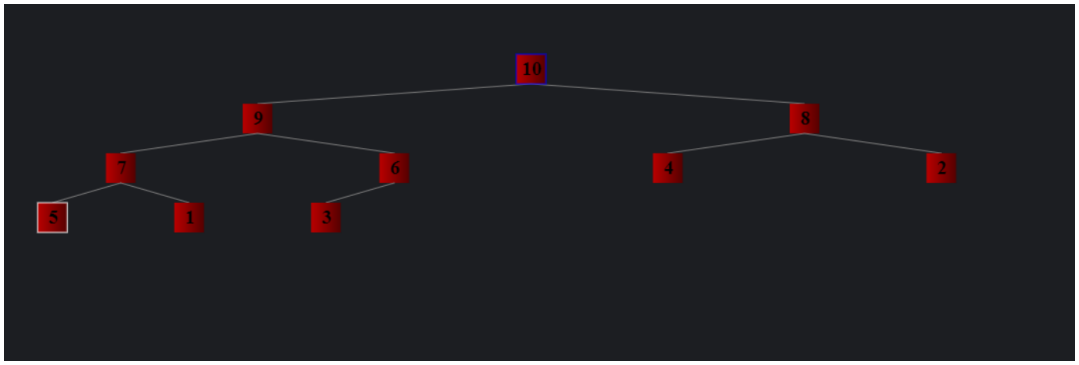
4b (30 p) Run heap sort by hand on this array, and show in detail in every step of the algorithm what calls to the heap are made and how the array and the heap change.

Solution: When we run heap sort on the array we get the sequence of heaps in Figures 3, 4, and 5. This particular example was worked out based on the code at `http://www.algostructure.com/sorting/heapsort.php` but using the code in the course material or lecture notes would of course also be OK (and would give very similar results).

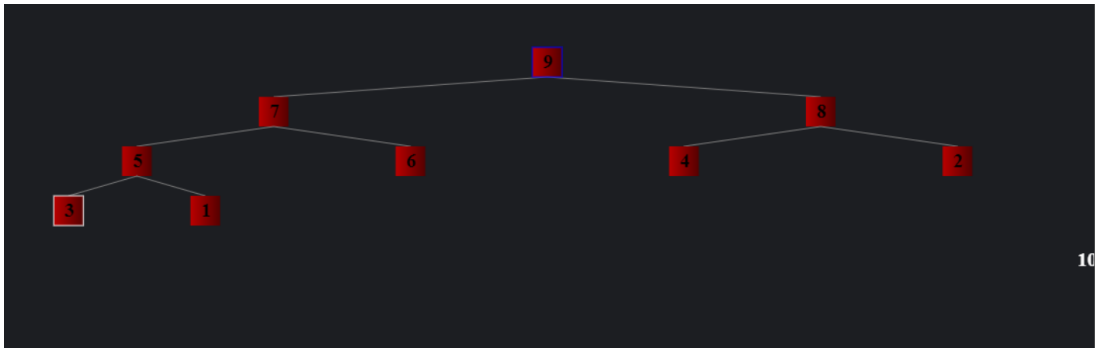
For a full score on this problem, you would need not just to illustrate how the heap changes, but also describe why. We do not do so here, but refer to the lecture notes where similar examples have been worked out in detail.

4c (20 p) Suppose that we are given another array B that is already sorted in increasing order. How fast do the merge sort and heap sort algorithms run in this case? Is any of them asymptotically faster than the other?

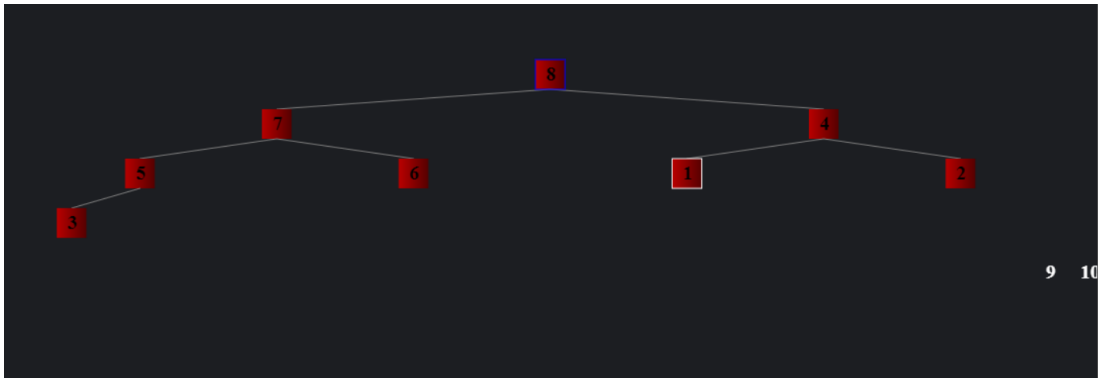
Solution: Merge sort always runs in time $\Theta(n \log n)$. One simple way of seeing this is that if we merge two lists $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$, then at least one element in every pair (a_i, b_i)



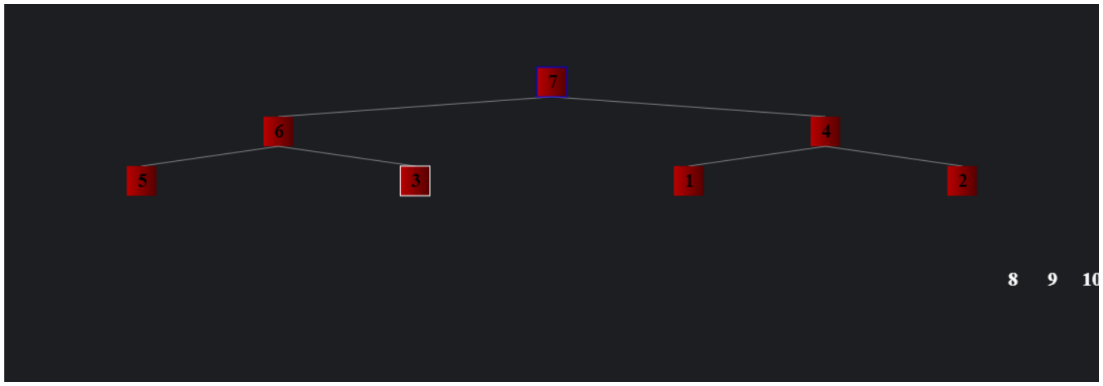
(a) Step 1



(b) Step 2

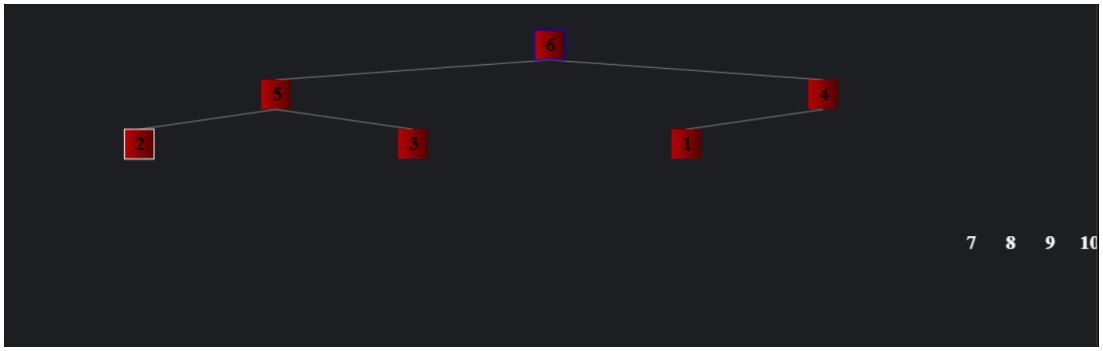


(c) Step 3

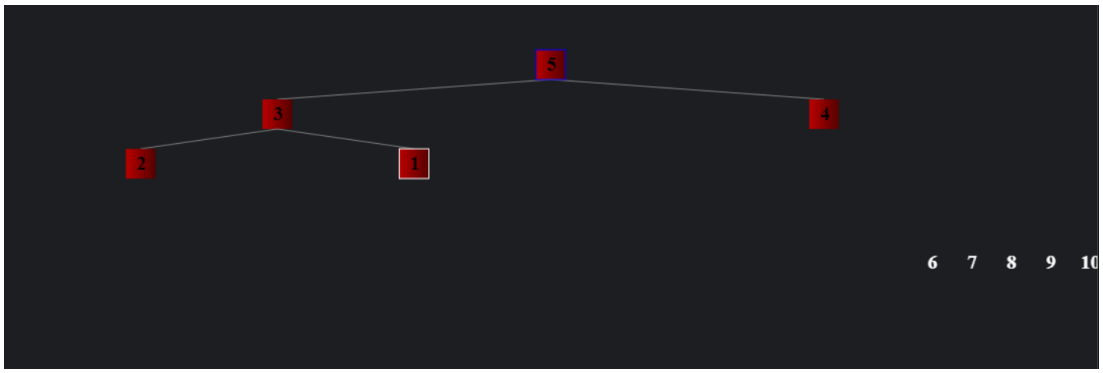


(d) Step 4

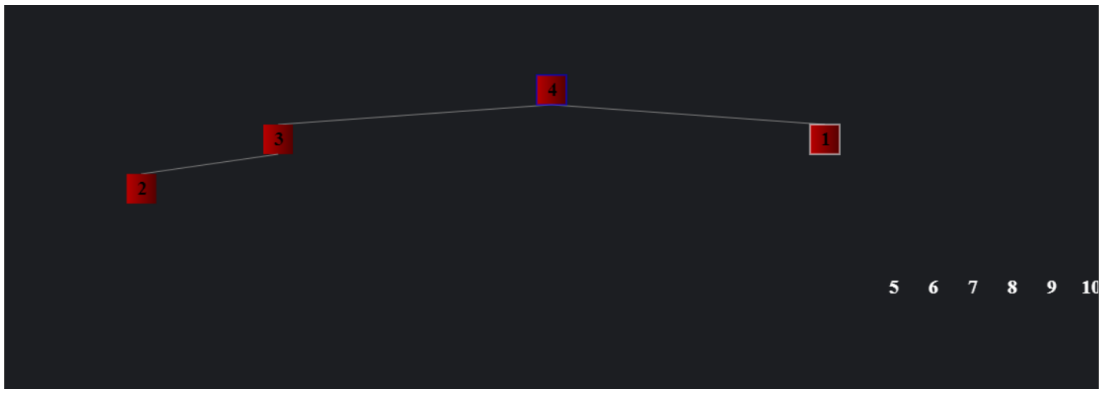
Figure 3: Heap sort: first figure.



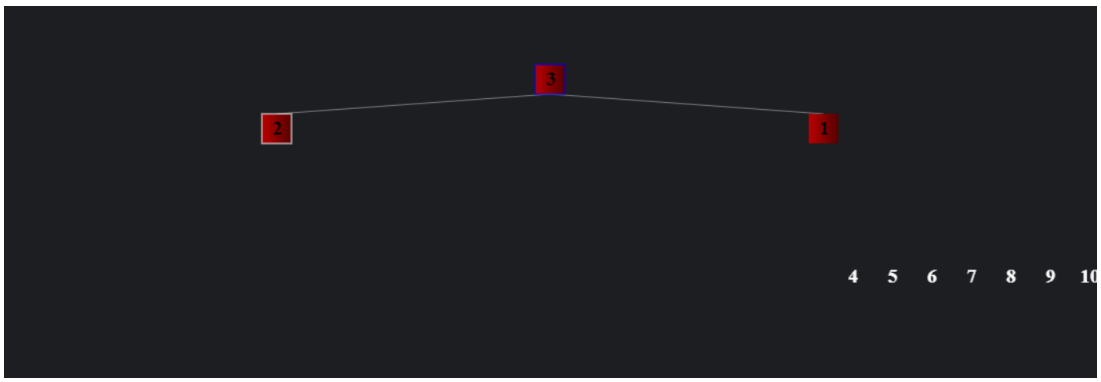
(a) Step 5



(b) Step 6



(c) Step 7



(d) Step 8

Figure 4: Heap sort: second figure.

(a) Step 9

(b) Step 10

Figure 5: Heap sort: third figure.

will be compared to an element in the other list. We will merge lists in (roughly) $\log n$ levels from sizes 1 up to $n/2$, and by the above argument we will have $\Omega(n)$ comparisons at each level.

Heapsort will also run in time $\Theta(n \log n)$, since we will need to restore the heap $n-1$ times, and the time needed to restore the heap can be $\Omega(\log n)$. It does not help that the array is already sorted.

Hence, the two sorting algorithms have the same (asymptotic) time complexity in this case.

4d (20 p) Suppose that we are given a third array C that is sorted in *decreasing* order, so that it needs to be reversed to be sorted in the order that we prefer, namely increasing. How fast do the merge sort and heap sort algorithms run in this case? Is any of them asymptotically faster?

Solution: Same answer as to the previous subproblem.

4e (10 p) Suppose that we build a max-heap from an 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 look at what the array from our max-heap in Problem 4c, it looks like $10, 9, 8, 7, 6, 4, 2, 5, 1, 3$. If we now 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.

5 (80 p) Decide for each of the propositional logic formulas below whether it is a tautology or a contradiction. If neither of these cases apply, then present a satisfying assignment for the formula. It is sufficient (and necessary) for a full score to justify all your answers by presenting truth tables for all the subformulas analogously to how we did it in class, but you are also encourage to *explain* why your answers are correct (and good explanations could compensate fully for minor slips in the truth tables).

(Note that logical not $\neg$ is assumed to bind harder than the binary connectives, but other than that all formulas are fully parenthesized for clarity. We write $\rightarrow$ for logical implication and $\leftrightarrow$ for equivalence.)

5a (20 p) $((p \rightarrow q) \rightarrow r) \rightarrow ((p \wedge q) \rightarrow r)$

Solution: For a full score the complete truth tables were needed. This is just mechanical work, and once the truth tables have been produced it is immediate to read off the correct answers. For Problem 5a we will both present the truth table and explain why the answer is correct, but for the other formulas we will skip the truth tables in these solution sketches and focus on the explanations.

The truth table for the formula in Problem 5a is as described below.

p	q	r	$p \rightarrow q$	$(p \rightarrow q) \rightarrow r$	$p \wedge q$	$(p \wedge q) \rightarrow r$	$((p \rightarrow q) \rightarrow r) \rightarrow ((p \wedge q) \rightarrow r)$
$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

From the rightmost column in this table we can see that the formula is a tautology. Let us now explain why this is so.

An implication $A \rightarrow B$ is false if and only if $A = \top$ and $B = \bot$. With this in mind, for our formula to be falsified it is required that $((p \wedge q) \rightarrow r)$ evaluates to false and $(p \rightarrow q) \rightarrow r$ to true. For the first of these formulas to be false we must have $r = \bot$, $p = \top$, $q = \top$. If we insert these values into the second formula we get $(\top \rightarrow \top) \rightarrow \bot$, which means that it will also be false. Thus, there is no way to falsify the whole formula, meaning that it is a tautology.

5b (20 p) $((p \rightarrow q) \wedge (r \rightarrow s)) \leftrightarrow ((p \wedge r) \rightarrow (q \wedge s))$

Solution: This formula has a falsifying assignment $p = \top, r = \bot, q = \bot, s = \top$ and a satisfying assignment $p = \top, r = \top, q = \top, s = \top$. Hence, it is neither a tautology nor a contradiction.

5c (20 p) $((p \wedge \neg r) \vee (q \wedge \neg r)) \rightarrow ((p \vee q) \rightarrow r)$

Solution: This is again neither a tautology nor a contradiction. A satisfying assignment is $p, q, r = \top$ and a falsifying assignment is $p, q = \top$, $r = \bot$.

5d (20 p) $(p \rightarrow (q \vee r)) \vee (\neg(\neg p \vee q) \wedge \neg r)$

Solution: If we negate the formula $p \to (q \vee r)$ and rewrite it using the rules we have learned in class, then we get

$$\begin{aligned}
\neg(p \to (q \vee r)) &\iff \neg(\neg p \vee (q \vee r)) && [\text{since } A \to B \iff \neg A \vee B] \\
&\iff \neg((\neg p \vee q) \vee r) && [\text{since } A \vee (B \vee C) \iff (A \vee B) \vee C] \\
&\iff \neg(\neg p \vee q) \wedge \neg r && [\text{since } \neg(A \vee B) \iff \neg A \wedge \neg B]
\end{aligned}$$

and this final subformula exactly matches the second part of the formula we are interested in. This means that the whole formula is eqivalent to a formula of the form $A \vee \neg A$, which is a tautology.

6 (100 p) Provide formal proofs of the following claims using proof techniques that we have learned during the course.

6a (20 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.

6b (20 p) For all $s \in \mathbb{N}$ and all $k > 0$ it holds that $1 + sk \leq (1+k)^s$.

Solution: First, we can note that the condition $k > 0$ in the problem statement is overly cautious—actually, $k > -1$ is enough, and we will see this in the proof below (although this requires some extra care in the argument which we do not need to worry about if we make k strictly positive). Second, since k could be any positive real number we cannot do inductive reasoning over k, but we can try to do induction over the integer s as follows.

Base case: For $s = 1$ we have $1 + k = (1+k)^1$ (and so the inequality $1 + k \leq (1+k)^1$ certainly also holds).

Induction step: Suppose that $1 + sk \leq (1+k)^s$ holds and consider $s+1$. We have

$$\begin{aligned}
(1+k)^{s+1} &= (1+k)(1+k)^s \\
&\geq (1+k)(1+sk) && [\text{by the induction hypothesis (and since } 1+k > 0)] \\
&= 1 + sk + k + sk^2 && [\text{multiplying out}] \\
&= 1 + (s+1)k + sk^2 && [\text{rearranging terms}] \\
&\geq 1 + (s+1)k && [\text{since } sk^2 \geq 0 \text{ as long as } s \geq 0]
\end{aligned}$$

which is the desired inequality.

The claim now follows by the induction principle.

Another way of establishing the inequality is to just do binomial expansion to compute

$$(1+k)^s = \binom{s}{0}1^s k^0 + \binom{s}{1}1^{s-1}k^1 + \ldots \binom{s}{s}1^0 k^s = 1 + sk + \cdots + k^s \geq 1 + ks$$

where the final inequality can be seen to hold since $k > 0$ (but note that the first proof presented above does not need this stronger assumption).

6c (30 p) For all $q \in \mathbb{N}$ it holds that $2^{4q+2} + 3^{q+2}$ is a multiple of 13.

Solution: This is very similar to a problem we did in class, and we will solve this one in the same way by arguing 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.

6d (30 p) Let $T_1 = 0$, $T_2 = 1$, and $T_i = T_{i-1} + T_{i-2}$ for $i \geq 3$. Prove that for all $i \geq 2$ it holds that

$$1 \leq \frac{T_{i+1}}{T_i} \leq 2 \ .$$

Solution: We prove this by induction over the numbers in the sequence, i.e., by induction over i. As a side note, the inequalities are actually sharp (i.e., $<$ rather than $\leq$) for $i \geq 4$, and this is easily seen from the proof that will follow below, but since the problem statement does not say anything about this there is no reason to comment on it in your solutions.

In order to make the proof clearer and avoid hidden assumption, we start by spelling out in full detail what induction hypotheses we will need.

Induction hypotheses: It holds for i that

1. $T_i > 0$ and $T_{i+1} > 0$;
2. $T_{i+1} \geq T_i$;
3. $T_i \leq T_{i+1} \leq 2 \cdot T_i$.

Note that we get the statement we need from this by dividing the inequalities in item 3 by T_i, which is in order since $T_i > 0$ by item 1.

Base case: For the base case $i = 2$ it is straightforward to verify for $T_2 = 1$ and $T_3 = T_2 + T_1 = 1$ that all of the induction hypotheses hold. (And yes, just writing like this in your exam solutions is perfectly in order, because you have said exactly what needs to be done, and the verification that what you say is true is fully mechanical and thus trivial.)

Induction step: Suppose that the induction hypotheses hold for i and consider $i+1$. Let us consider the hypotheses one by one.

1. Since T_i and T_{i+1} are both strictly positive by the induction hypotheses, it holds that $T_{i+2} = T_{i+1} + T_i$ is the sum of two strictly positive numbers and hence also strictly positive.
2. Since $T_i > 0$ by the induction hypotheses, we have $T_{i+2} = T_{i+1} + T_i > T_{i+1}$.
3. Since $T_{i+1} \geq T_i$ by the induction hypotheses, we can plug this into the recurrence relation to get $T_{i+2} = T_{i+1} + T_i \leq 2 \cdot T_{i+1}$. (And note that since $T_4 > T_3$ we could get a strict inequality here if we wanted to.)

The inequalities follow by the induction principle.

7 (60 p) In this problem we focus on relations. In particular, suppose that $A = \{e_1, e_2, \ldots, e_9, e_{10}\}$ is a set of 10 elements and consider the relation R on A represented by the matrix

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

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

7a (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 describe 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

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

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

7b (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 describe 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

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

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

7c (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 7b, 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, it is true for any relation R that the relations S_1 and S_2 will be the same.

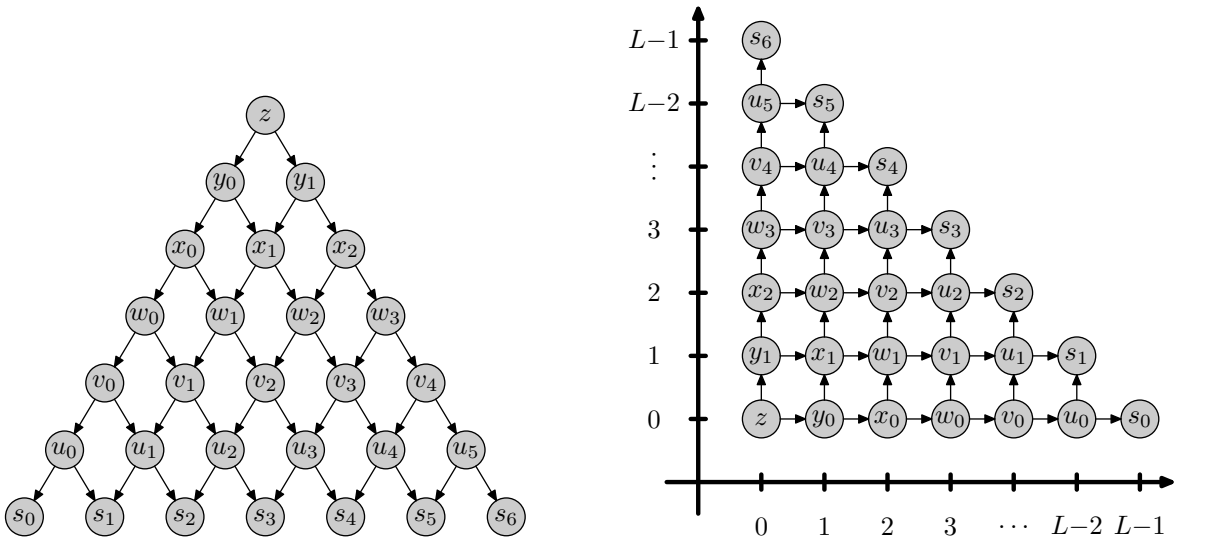
8 (100 p) Consider a directed graph that consists of L layers numbered from 0 up to $L - 1$, with $i + 1$ vertices in every layer i numbered from 0 to i, and with outgoing edges from vertex j in layer i to vertices j and $j + 1$ in layer $i + 1$. See Figure 6a for an illustration of this graph with 7 layers. We can agree to call this graph a *lattice graph* (because it can be obtained from a fragment of the integer lattice in 2 dimensions as shown in Figure 6b, but this is actually completely irrelevant to this problem).

Suppose we start in the unique source vertex on level 0 and walk along edges in the graph, flipping a fair coin at every vertex to decide whether to go left or right, until we reach one of the sinks in the last layer. For instance, in Figure 6a the walk "left–left–right–left–right–right" would visit vertices z, y_0, x_0, w_1, v_1, u_2, and end in s_3.

8a (40 p) For every vertex s_i in the lattice graph in Figure 6a, calculate the probability that such a walk ends in vertex s_i. Which vertex is the walk most likely to end up in?

8b (60 p) For a lattice graph with L layers, where $L \geq 2$ is a positive integer, calculate the probability that a walk as described above ends in vertex j in layer $L - 1$ for $j = 0, 1, \ldots, L - 1$.

Hint: Use the fact that all walks are equally likely to turn this into a counting problem.



(a) Lattice graph with 7 layers.

(b) Graph as connected points in 2D integer lattice.

Figure 6: Example graph for Problem 8.

Solution: Clearly, Problem 8a is a special case of Problem 8b, and here we will focus on the latter problem.

Using the hint, since a fair coin is being flipped we deduce that the probability of any particular walk is the same and is equal to $\left(\frac{1}{2}\right)^{L-1}$. In order to calculate the probability of reaching a certain leaf, it is sufficient to count the number of walks ending in that leaf. We can identify each walk with a binary sequence of length $L-1$, where 1 means going right and 0 means going left, say, so that the walk "left–left–right–left–right–right" in Figure 6a visiting vertices z, y_0, x_0, w_1, v_1, u_2, and s_3 is encoded as the sequence 001011.

Thinking a bit more, we realize that which leaf the walk ends up in is determined by the number of right turns in the walk, regardless of when these turns happen. Looking again at Figure 6a, any walk making exactly 3 right turns will end up in s_3. This means that the probability of reaching leaf j for $j = 0, 1, \ldots, L-1$ is proportional to the number of walks with j right turns, or the number of binary strings of length $L-1$ containing exactly j ones. But this number is nothing other than the binomial coefficient $\binom{L-1}{j}$. Thus, the probability of reaching leaf j can be written as

$$\Pr[\text{reaching leaf } j] = \binom{L-1}{j}\left(\frac{1}{2}\right)^{L-1} .$$

9 (80 p) Consider the graph in Figure 7. We assume that this graph is represented in adjacency list format, with the neighbours in each adjacency list sorted in lexicographic order (i.e., in the order $a, b, c, d, \ldots$).

9a (30 p) Run the code for depth-first search by hand on this graph, starting in the vertex a. Describe in every recursive call which neighbours are being considered and how they are dealt with. Show the spanning tree produced by the search.

Solution: See Figure 8 for an illustration of the algorithm execution described below (with every node marked by the times it was visited and finished) and the resulting spanning tree (redrawn separately in red for clarity).

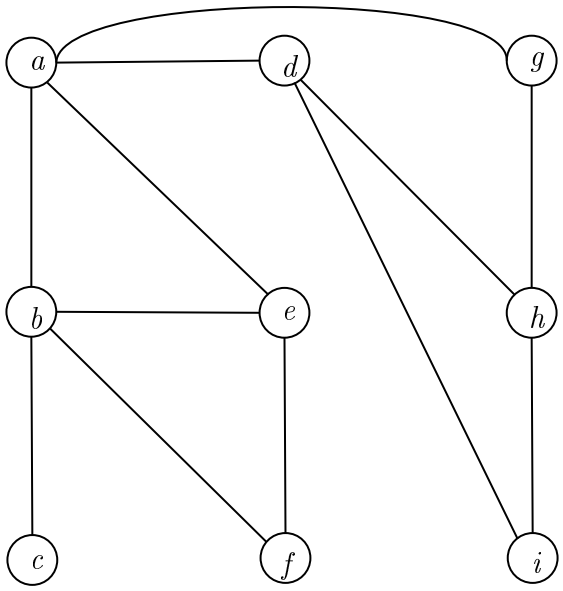


Figure 7: Undirected unweighted graph for graph traversals in Problem 9.

1. We start in vertex a, which we mark as visited. Since the neighbours in each adjacency list are sorted in lexicographic order, we first neighbour of a that we look at is b.

2. Vertex b is not visited, so we make a recursive call for b, and add the edge (a, b) to the spanning tree. The first neighbour of b is a, which is visited.

3. The next neighbour c of b is not visited, so we make a recursive call for c and add the edge (b, c) to the spanning tree. The only neighbour of c is b, which is already visited, so the recursive call returns.

4. We are now back in the call made for b. The next neighbour e of b is not visited, so we make a recursive call for e and add the edge (b, e) to the spanning tree. The first neighbours of e are a and b, which are both already visited.

5. The next neighbour f of e is not visited, so we make a recursive call for f and add the edge (e, f) to the spanning tree. The neighbours b and e of f are both visited, so the recursive call returns.

6. We are now back in the call made for e, but since all neighbours have now been processed, this call returns.

7. This brings us back to the call made for b, and since all neighbours of b have also been processed, this call returns.

8. At this point, we have return to the very first DFS call made, namely for a. The next neighbour of a after b is d, which has not been visited, so we visit d and add the edge (a, d) to the tree. The first neighbour a of d has been visited.

9. The next neighbour of d is h, which we see for the first time and so visit, adding (d, h) to the tree. The first neighbour of h is d, which is where we came from.

10. The second neighbour of h is g, which we see for the first time and so visit, adding (g, h) to the tree. The neighbours of g are a and h, which have both been visited, so the recursive call to g immediately returns.

11. The third and final neighbour of h is i, which we visit, adding the edge (h, i) to the spanning tree.

12. Now all vertices have been visited, so in what remains the algorithm will finish the recursive calls to i, h, d, and a (in this order) by discovering for each vertex that there are no non-visited vertices left.

9b (30 p) Run the code for breadth-first search by hand on this graph, also starting in the vertex a. Describe for every vertex which neighbours are being considered and how they are dealt with, and how the queue changes. Show the spanning tree produced by the search.

Solution: See Figure 9 for an illustration of the algorithm execution described below and the resulting spanning tree.

1. At the start, the queue contains only the vertex a.

2. We dequeue a and add its neighbours b, d, e, and g to the queue in this order, as well as the edges (a, b), (a, d), (a, e), and (a, g), to the spanning tree, since none of these vertices had been seen before. The queue now looks like (b, d, e, g).

3. We dequeue b. Neighbour a is already marked, but c is not and so is enqueued. Neighbour e is also marked, but f is not and so is enqueued. The edges (b, c) and (b, f) are added to the spanning tree. The queue now looks like (d, e, g, c, f).

4. We dequeue d. Neighbour a is already marked, but h and i are new vertices. We therefore enqueue them and add the edges (d, h) and (d, i) to the spanning tree. The queue now looks like (e, g, c, f, h, i).

5. We dequeue e. All neighbours are already marked, so no new vertices are enqueued and the queue shrinks to (g, c, f, h, i).

6. We dequeue g. Again all neighbours are marked, so no new vertices are enqueued and the queue shrinks to (c, f, h, i).

7. The algorithm will continue like this until the queue is empty, because all vertices have already been discovered and so no new vertices can be added. (This was true already after processing d, and it would also have been in order to point this out immediately after this step.)

9c (20 p) Suppose that the graph in Figure 7 is modified to give every edge weight 1, and that we run Dijkstra's algorithm starting in vertex a. How does the tree computed by Dijkstra's algorithm compare to that produced by DFS? What about BFS?

Solution: If all edges have the same weight 1, then Dijkstra's algorithm produces the same result as breadth-first search. Just as the BFS trees and DFS trees can be very different, there is no particular relation between the DFS tree and what Dijkstra's algorithm would produce in this case.

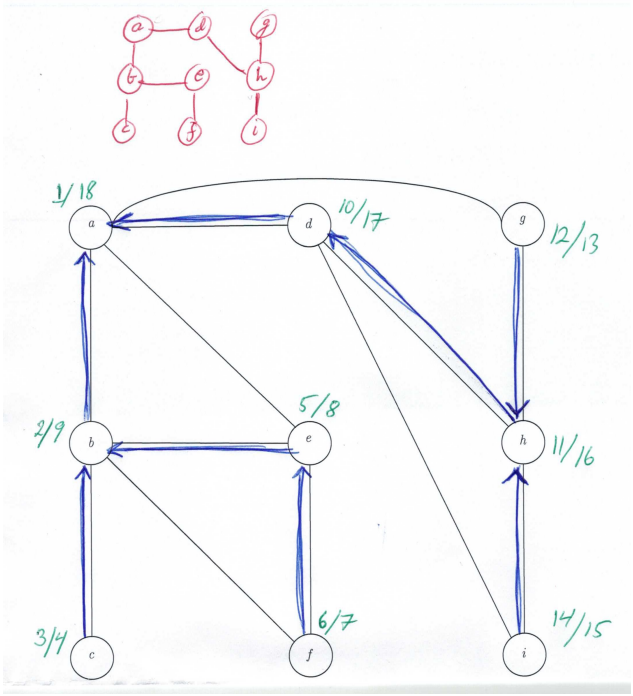


Figure 8: Depth-first search for graph in Figure 7.

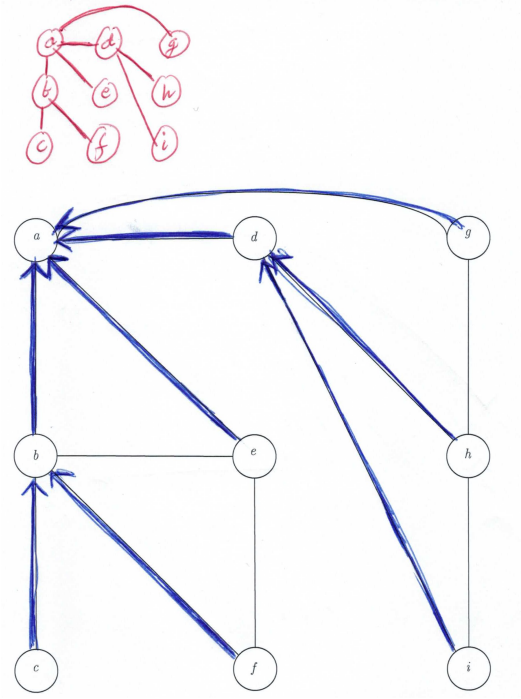


Figure 9: Breadth-first search for graph in Figure 7.

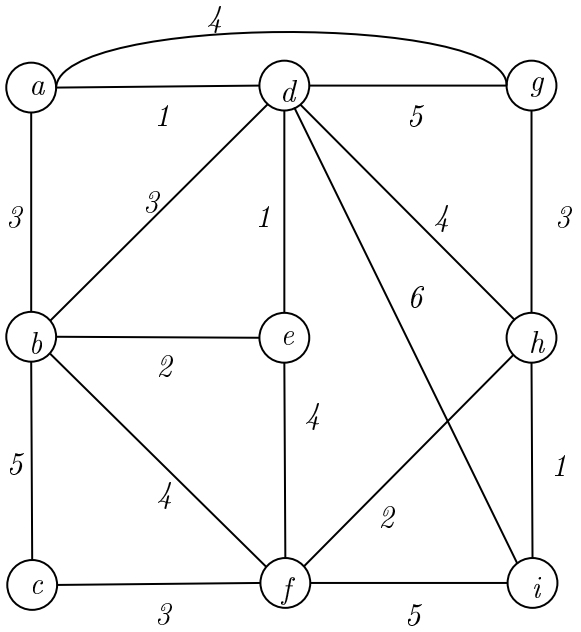


Figure 10: Undirected weighted graph for minimum spanning trees in Problem 10 and shortest paths in Problem 11.

10 (110 p) Consider the graph in Figure 10. We assume that this graph is represented in adjacency list format, with the neighbours in each adjacency list sorted in lexicographic order.

10a (30 p) Generate a minimum spanning tree by running Prim's algorithm by hand on this graph, starting in the vertex a. Use a heap for the priority queue implementation. Describe for every dequeueing operation which neighbours are being considered and how they are dealt with, and show how the heap changes. Also show the tree produced by the algorithm.

Solution: The execution of Prim's algorithm is illustrated on the graph in Figure 11. We do not describe in this solution sketch the details of how the heap changes. At the outset, the vertex a has key 0 and all other vertices have key ∞. In what follows, if vertex v has key value k, we will use the notation (v, k) for this for brevity.

1. Dequeue a. The keys of the neighbours of a are updated to $(b, 3)$, $(d, 1)$, $(g, 4)$, after which d is at the front of the queue.

2. Dequeue d and add edge (a, d) to the spanning tree. The keys of the neighbours of d are updated to $(e, 1)$, $(h, 4)$, $(i, 6)$. Neighbours b and g already have values that are not larger than the weights of the edges to d, and vertex a has already been processed. Now e is at the front of the queue.

3. Dequeue e and add edge (d, e) to the spanning tree. The keys of the neighbours are updated to $(b, 2)$, $(f, 4)$. Now b is at the front of the queue.

4. Dequeue b and add edge (b, e) to the spanning tree. The key of c is updated to 5. All other neighbours of b are already processed or have at least as good keys already. Now either f or g is at the front of the queue—let us say it is f, just because f comes before g in alphabetical order.

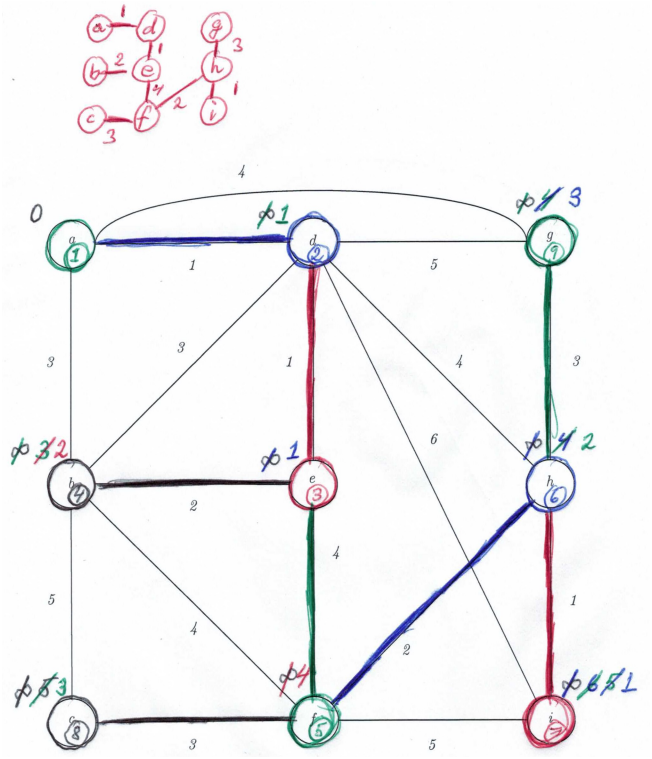


Figure 11: Minimum spanning tree for graph in Figure 10 using Prim's algorithm.

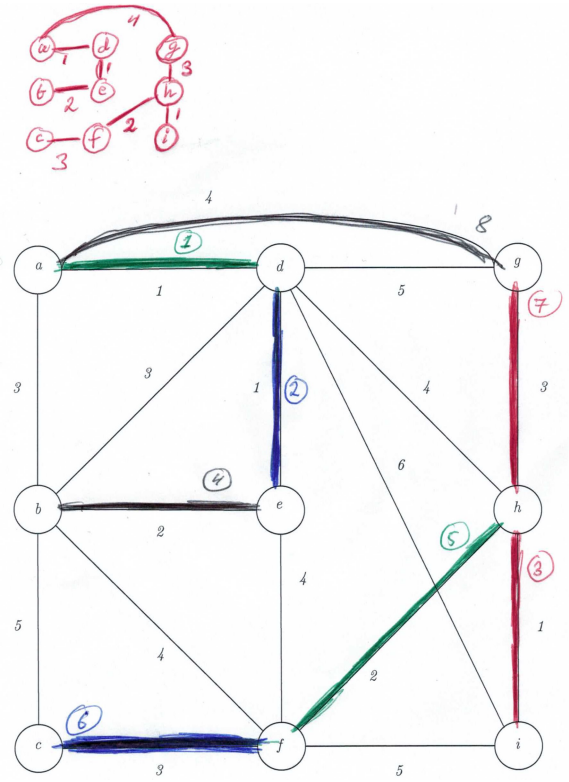


Figure 12: Minimum spanning tree for graph in Figure 10 using Kruskal's algorithm.

5. Dequeue f and add edge (e, f) to the spanning tree. The neighbour key updates that are made in this step are $(c, 3)$, $(h, 2)$, $(i, 5)$, and g is no longer next in queue, since both h and c sneaked ahead.

6. Dequeue h and add edge (f, h) to the spanning tree. The neighbour key updates that are made in this step are $(g, 3)$, $(i, 1)$, and so now i is next in queue.

7. Dequeue i and add edge (h, i). There are no key updates, and either c or g is at the front of the queue—let us say that g loses out again.

8. Dequeue c and add edge (c, f). There are no key updates, g is the only vertex left in the queue.

9. Dequeue g and add edge (g, h) to the minimum spanning tree, which is now complete.

As a side note, we remark that the website `https://visualgo.net/en/mst` provides nice visualizations for both Kruskal's and Prim's algorithm.

10b (30 p) Generate a minimum spanning tree by running Kruskal's algorithm by hand on this graph. Assume that edges of the same weight are sorted in lexicographic order (so that we would have (a, b) coming before (a, d), which would in turn come before (b, a) for hypothetical edges of equal weight). Describe how the forest changes at each step (but you do not need to describe in detail how the set operations are implemented). Also show the final tree produced by the algorithm.

Solution: We illustrate the execution of Kruskal's algorithm in Figure 12. After sorting, the edges will be processed in the following order:

Weight 1: (a, d), (d, e), (h, i).

Weight 2: (b, e), (f, h).

Weight 3: (a, b), (b, d), (c, f), (g, h).

Weight 4: (a, g), (b, f), (d, h), (e, f).

Weight 5: (b, c), (d, g), (f, i).

Weight 6: (d, i).

The algorithm now does the following:

1. Looking first at edges of weight 1, the edge (a, d) creates a forest $\{a, d\}$ and is added.

2. The edge (d, e) enlarges this forest to $\{a, d, e\}$ and is added.

3. The edge (h, i) creates a forest $\{h, i\}$ and is added.

4. Moving on to edges of weight 2, the edge (b, e) grows the forest $\{a, d, e\}$ further to $\{a, b, d, e\}$ and is added.

5. The edge (f, h) grows the forest $\{h, i\}$ to $\{f, h, i\}$ and is added.

6. For the edges of weight 3, we see that (a, b) and (b, d) would both create cycles in the forest $\{a, d, e\}$, so they are discarded. The edge (c, f) grows the forest $\{f, h, i\}$ to $\{c, f, h, i\}$ and is added.

7. The edge (g, h) grows the forest $\{c, f, h, i\}$ to $\{c, f, g, h, i\}$ and is added.

8. For the weight-4 edges, according to our sorting order the edge (a, g) comes first, and is added since it creates a spanning tree for the full graph. Since we know we have now added the right number of edges, we can terminate without considering any further edges.

We note that in the last step we could also have added the edge (e, f) instead, if the edges of weight 4 would have been sorted differently, and this would have given us the minimum spanning tree in Figure 11. It would also have been possible to pick either of the two other weight-4 edges (b, f) or (d, h) to get other minimum spanning trees.

10c (10 p) Suppose that some vertex v with several neighbours in a graph G has a unique neighbour u such that the edge (u, v) has strictly smaller weight than any other edge incident to v. Is it true that the edge (u, v) must be included in any minimum spanning tree? Prove this or give a simple counter-example.

Solution: Yes, any MST has to include the edge (u, v), since this is the unique light edge in a cut with vertex v on one side and the rest of the graph on the other side. If there were an MST without the edge (u, v), then adding that edge to the MST would create a cycle, from which we could remove a heavier edge and get another spanning tree for the graph. This contradicts that the tree we started with was an MST.

10d (20 p) Suppose that some vertex v with several neighbours in a graph G has a unique neighbour u such that the edge (u, v) has strictly larger weight than any other edge incident to v. Is it true that the edge (u, v) can never be included in any minimum spanning tree? Prove this or give a simple counter-example.

Solution: No, this is not true. Consider the case when u has only one neighbour and this neighbour is v. Then the edge (u, v) has to be included in any MST.

10e (20 p) Suppose that T is a minimum spanning tree for a weighted, undirected graph G. Modify G by adding some constant $c \in \mathbb{R}^+$ to all edge weights. Is T still a minimum spanning tree? Prove this or give a simple counter-example.

Solution: If the spanning tree has M edges, then after the edge weight increase the new tree has a total weight that increased by an amount $M \cdot c$. But any spanning tree for a graph G will have the same number of edges, and hence the weight increase will be exactly the same for all spanning trees. This means, in particular, that all MSTs before the weight increase remain MSTs also after the weight increase.

11 (60 p) Consider again the graph in Figure 10 with the same assumptions.

11a (40 p) Run Dijkstra's algorithm by hand on this graph, starting in the vertex a. Use a heap for the priority queue implementation. Describe for every vertex which neighbours are being considered and how they are dealt with, and show how the heap changes. Also show the tree produced by the algorithm.

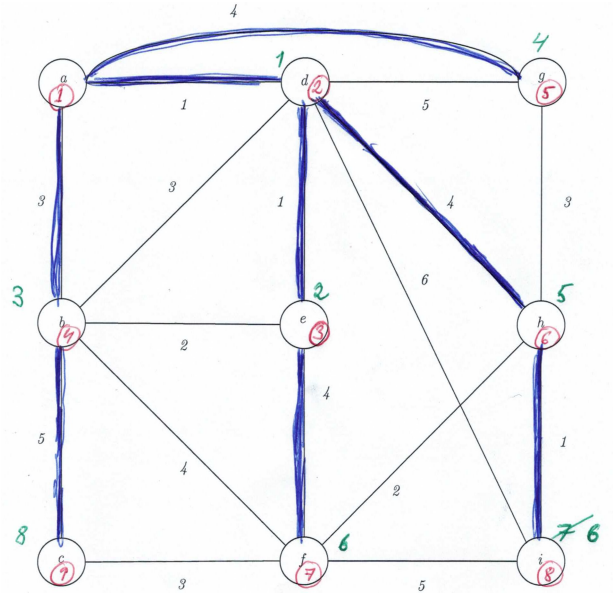


Figure 13: Shortest paths computed by Dijkstra's algorithm on the graph in Figure 10.

Solution: The execution of Dijkstra's algorithm is illustrated on the graph in Figure 13. We do not describe in this solution sketch the details of how the heap changes. At the outset, the vertex a has key 0 and all other vertices have key ∞. We will again use the notation (v, k) when vertex v has key value k.

1. Dequeue a and relax the neighbours to get $(b, 3)$ $(d, 1)$, $(g, 4)$; leaving d at the front of the priority queue.
2. Dequeue d, add the edge (a, d), and relax the neighbours to get $(e, 2)$, $(h, 5)$, $(i, 7)$. Note that a, b, and g are not affected by the relaxation since their key values do not decrease. Now e is first in line.
3. Dequeue e, add the edge (d, e), and relax the neighbours to get $(f, 6)$. Vertices b and d are not affected, and now b is at the front of the queue.
4. Dequeue b, add the edge (a, b), and relax the neighbours to get $(c, 8)$. No other neighbours of b are affected by the relax operations. Now g is first in line.
5. Dequeue g and add the edge (a, g). No neighbours of g are affected by the relax operations. Now h is first in line.
6. Dequeue h, add the edge (d, h), and relax the neighbours to get $(i, 6)$. No other neighbours of h are affected by the relax operations. Now either f or i is first in the queue.
7. In the last two steps, the edge (e, f) is added for f and the edge (h, i) is added to reach i. This concludes the execution of the algorithm.

We remark that there is a nice simulation for running Dijkstra's algorithm at `https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html`.

11b (20 p) If Dijkstra's algorithm is run on an undirected graph, is it true that the spanning tree produced is a minimum spanning tree? Prove this or give a simple counter-example.

Solution: No, this is not true, and the spanning tree produced in our solution of Problem 11a shows this. (For instance, we can decrease the weight of this tree by replacing edge (a, b) by edge (b, e), or by replacing edge (d, h) by edge (g, h).)

12 (120 p) In this problem, we want to understand the languages generated by specified regular expressions and context-free grammars.

12a (5 p per correct answer below) Which of the words below belong to the language generated by the regular expression $(b(ab)^* a) \,|\, (b^* ab^* ab^*)$? Motivate briefly your answers.

1. *babababa*
2. *babbabbb*
3. *ba*
4. *babaabaaab*
5. *babbaabbbba*
6. *babba*

Solution: The regular expression accepts words that either have one b followed by arbitrary number of (ab) ending with a or that have two occurrences of a with an arbitrary number of b before, in between, or after. With this in mind, we get the following answers:

1. $babababa = b(ab)^3a$ ✓
2. $babbabbb = ba(b)^2a(b)^3$ ✓
3. $ba = b(ab)^0a$ ✓
4. *babaabaaab* ✗ (Letters a and b do not alternate, ruling out the first alternative, and for the second alternative there are too many occurrences of a.)
5. *babbaabbbba* ✗ (Same reason again.)
6. $babba = ba(b)^2a(b)^0$ ✓

12b (30 p per correct answer below) Consider the following context-free grammars, where a, b, c, d are terminals, A, B, S are non-terminals, and S is the starting symbol.

Grammar 1:

$$S \to abS \tag{1a}$$
$$S \to B \tag{1b}$$
$$B \to aB \tag{1c}$$
$$B \to cB \tag{1d}$$
$$B \to dB \tag{1e}$$
$$B \to \tag{1f}$$

Grammar 2:

$$S \to AbS \tag{2a}$$
$$S \to \tag{2b}$$
$$A \to aAa \tag{2c}$$
$$A \to \tag{2d}$$

Grammar 3:

$$S \to aSa \tag{3a}$$
$$S \to bS \tag{3b}$$
$$S \to \tag{3c}$$

Which of these grammars generate regular languages? For each grammar, write a regular expression that generates the same language, or argue why the language generated by the grammar is not regular.

Solution: Grammar 1 generates strings that contain zero or more occurrences of ab follows by zero or more occurrences of (any alternation of) characters a, c, or d. This languages is captured by the regular expression $(ab)^*(a|c|d)^*$.

Grammar 2 generates zero or more substrings of the format where every substring consists of an even (possibly zero) number of occurrences of a followed by exactly one b. This corresponds to the regular expression $\big((aa)^*b\big)^*$.

The language generated by Grammar 3 is not regular. To see this, note that strings of the form a^nba^n are in the language while strings of the form a^nba^{n+1} are not. In order to distinguish between the two, the regular expression would need to do counting, but as mentioned in Mogesen's notes and during the lectures this is something that regular expressions cannot do. (You do not have to prove this to get full credits for this problem—referring to what is said in the notes or what we covered in class is enough.)

13 (120 p) In this problem, we want to write regular expressions and context-free grammars generating specified languages.

13a (30 p) Write a regular expression for the language consisting of all finite (possibly empty) bit strings (i.e., over the alphabet $\{0, 1\}$) that contain an even number of 0s. Examples of such strings are ε (the empty string), 00, 1010001, and 111, whereas 10 and 01010 do not qualify for membership.

Solution: The string can start with a sequence of 1s, leading up to the first 0, if any. If there is a 0, then it should be paired with another occurrence of 0, and in between them and after them we could have any number of 1s (including none). We can write this down as the regular expression $1^*(01^*01^*)^*$. (Other solutions are also possible, of course.)

13b (50 p) Write a regular expression for the language consisting of all finite, non-empty bit strings that contain no two consecutive 1s. Examples of strings in this language are 0000, 01010, and 1001, while 111, 00110, and ε do not qualify. For partial credit (if your regular expression is wrong or missing), argue why this language is clearly regular.

Solution: To see that this language is regular, it is sufficient to build a deterministic finite automaton that recognizes it, such as the DFA in Figure 14.

Suppose the string both starts and ends with a 0. Such strings are captured by the expression $00^*(100^*)^*$. To allow the string to also end in a 1, we can expand the expression slightly to $00^*(100^*)^*(1|\varepsilon)$.

If the string instead starts and ends with a 1, we can describe it with the expression $1(00^*1)^*$. If we want to allow the string to end with 0 in this case, we can expand the expression to $1(00^*10^*)^*$.

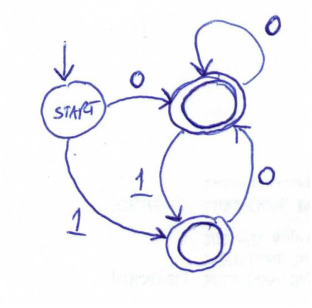


Figure 14: DFA proving regularity of language in Problem 13b.

Since the string will start with either 0 of 1, taking the alternative of these two expressions to get $\big(00^*(100^*)^*(1|\varepsilon)\big)|\big(1(00^*10^*)^*\big)$ is one possible solution.

We note that a much more elegant expression, which gets the language almost right, is $(1|\varepsilon)(00^*1)^*0^*$, but this regular expression also accepts the empty string, which is not allowed according to the description in the problem statement.

13c (40 p) Give a context-free grammar for the language $\left\{a^m b^n c^{m+n} \,\middle|\, m, n \in \mathbb{N}\right\}$. Examples of strings in this language are *aacc* and *abbccc*, whereas *abc* and *abccc* do not make the cut.

Solution: Strings in this language should start with zero or more occurrences of a, with every occurrence being matched by a c at the end. At some point the string should switch to zero or more occurrences of b, again with every occurrence being matched by a c at the end, and no a can appear after the first b. (There is a slight ambiguity here in that we have to decide whether 0 is a natural number or not, but the example strings make clear that this is considered to be the case.)

A language as described above can be generated by, e.g., the following grammar:

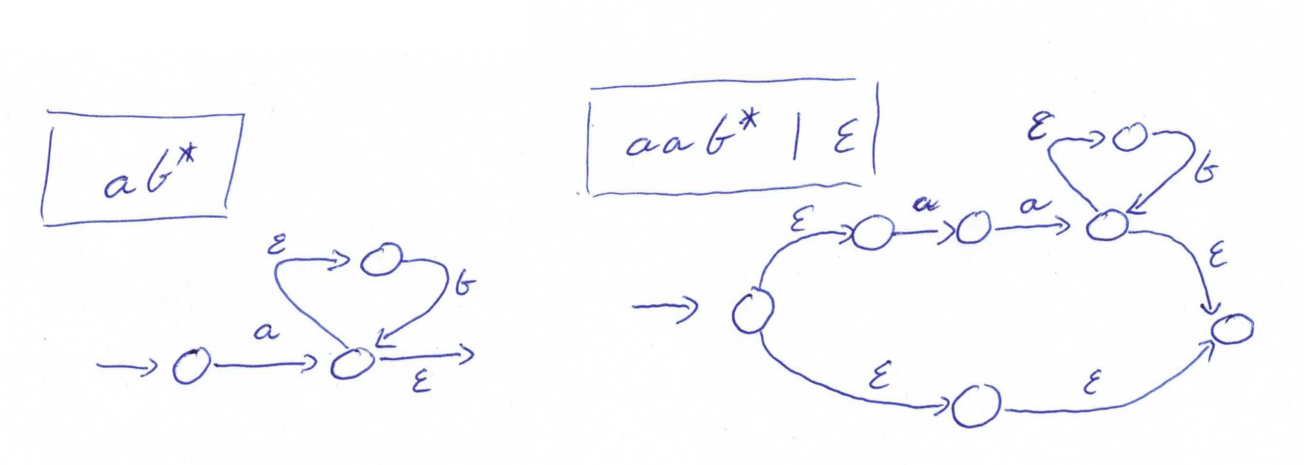
$$S \to aSc$$
$$S \to B$$
$$B \to bBc$$
$$B \to \varepsilon$$

14 (120+ p) Consider the regular expression $b^*(ab^*(aab^*|\varepsilon)cb^*)^*$

14a (60 p) Translate this regular expression to a nondeterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain which part of the regular expression corresponds to which part of the NFA.

Solution: We follow the procedure outlined in Section 1.3 in Mogesen's notes. Two NFA fragments with corresponding regular (sub)expressions are shown in Figure 15. The full translation of the regular expression to a nondeterministic finite automaton is as in Figure 16, where we have added numbers to the states to be able to refer to them in the next subproblem.

14b (60 p) Translate the nondeterministic finite automaton to a deterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain how you perform the subset construction, so that it is possible to follow your line of reasoning.



(a) NFA fragment for ab^*.

(b) NFA fragment for $aab^*|\varepsilon$.

Figure 15: NFA fragments in translation of regular expression in Problem 14.

Solution: We follow Algorithm 1.3 in Section 1.5.2 of Mogesen's notes. We will use calligraphic letters $\mathcal{A}, \mathcal{B}, \ldots$ to refer to the constructed states in the deterministic finite automaton. Referring in what follows to the numbered states in the NFA in Figure 16, the starting state is

$$\varepsilon\text{-closure}(1) = \{1, 2, 3, 4, 17\} = \mathcal{A}$$

(where we recall that the *ε-closure* of a set of states S is any state that can be reached from some state in S via zero or more ε-transitions). The possible transitions from $\mathcal{A}$ on characters a, b, and c are

$$\begin{aligned}
&\text{move}(\mathcal{A}, a) = \varepsilon\text{-closure}(5) = \{5, 6, 7, 8, 12, 13, 14\} = \mathcal{B} && [\textbf{new state}]\\
&\text{move}(\mathcal{A}, b) = \varepsilon\text{-closure}(1) = \mathcal{A}\\
&\text{move}(\mathcal{A}, c) = \emptyset && [\text{no transition on } c \text{ possible}]
\end{aligned}$$

We consider next the new state $\mathcal{B}$, for which we get the transitions

$$\begin{aligned}
&\text{move}(\mathcal{B}, a) = \varepsilon\text{-closure}(9) = \{9\} = \mathcal{C} && [\textbf{new state}]\\
&\text{move}(\mathcal{B}, b) = \varepsilon\text{-closure}(5) = \mathcal{B}\\
&\text{move}(\mathcal{B}, c) = \varepsilon\text{-closure}(15) = \{3, 4, 15, 16, 17\} = \mathcal{D} && [\textbf{new state}]
\end{aligned}$$

Moving on to state $\mathcal{C}$ we obtain

$$\begin{aligned}
&\text{move}(\mathcal{C}, a) = \varepsilon\text{-closure}(10) = \{10, 11, 13, 14\} = \mathcal{E} && [\textbf{new state}]\\
&\text{move}(\mathcal{C}, b) = \emptyset && [\text{no transition on } b \text{ possible}]\\
&\text{move}(\mathcal{C}, c) = \emptyset && [\text{no transition on } c \text{ possible}]
\end{aligned}$$

and for state $\mathcal{D}$ we derive

$$\begin{aligned}
&\text{move}(\mathcal{D}, a) = \varepsilon\text{-closure}(5) = \mathcal{B}\\
&\text{move}(\mathcal{D}, b) = \varepsilon\text{-closure}(15) = \mathcal{D}\\
&\text{move}(\mathcal{D}, c) = \emptyset && [\text{no transition on } c \text{ possible}]
\end{aligned}$$

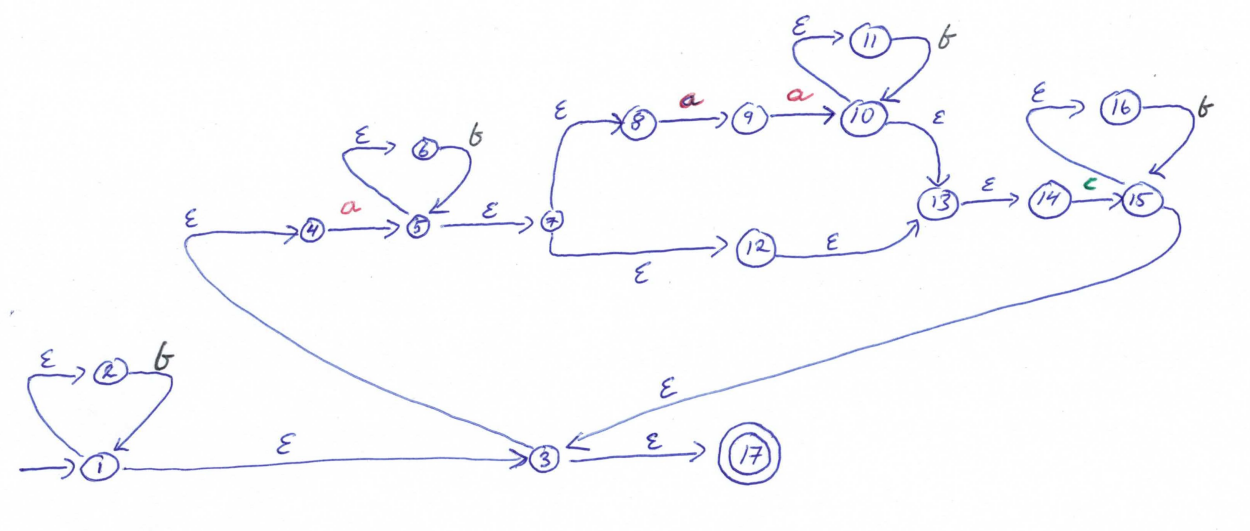


Figure 16: Full translation to NFA of regular expression in Problem 14.

Finally, for state $\mathcal{E}$ we have

$$\begin{aligned} &\text{move}(\mathcal{E}, a) = \emptyset && \text{[no transition on } a \text{ possible]} \\ &\text{move}(\mathcal{E}, b) = \varepsilon\text{-closure}(10) = \mathcal{E} \\ &\text{move}(\mathcal{E}, c) = \varepsilon\text{-closure}(15) = \mathcal{D} \end{aligned}$$

and since we have now considered all possible transitions from all states in our constructed DFA we are done. The new states containing the accepting state 17 in the NFA are $\mathcal{A}$ and $\mathcal{D}$, so these are the accepting states in our DFA. See Figure 17 for an illustration of the constructed automaton.

14c *Bonus problem (worth 20 p extra):* How small a DFA can you produce that accepts precisely the language generated by the regular expression above? (Note that that you can solve this problem even if you did not solve the other subproblems above.)

Solution: Looking at Figure 17, it is not hard to see that the states $\mathcal{A}$ and $\mathcal{D}$ have the same outgoing transitions. We can therefore merge these two states to get a smaller DFA as in Figure 18. Although we will not attempt a formal argument here, it is intuitively clear that this latter DFA is of optimal size. This is so since the automaton has to be able to count the number of occurrences of a that it has seen since the last occurrence of c, and this number can be in the range from 0 to 3.

We remark that it is not necessary here to go the formal route by first construction an NFA from the regular expression and then converting this NFA to a DFA. Just by studying the regular expression and understanding what it means, it is possible to write down the DFA in Figure 17 directly (and indeed, this was how the main instructor first did it).

15 (170 p) In this problem we want to construct parsers for context-free languages. We assume that (,), [,], +, *, and **num** are tokens return by a lexer (i.e., from the point of view of the parser these are terminals in the alphabet).

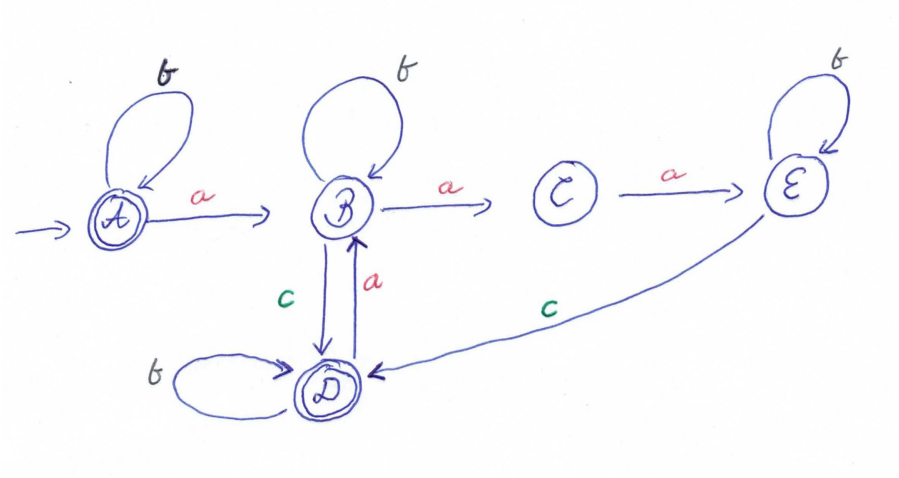


Figure 17: Conversion of NFA in Figure 16 to DFA.

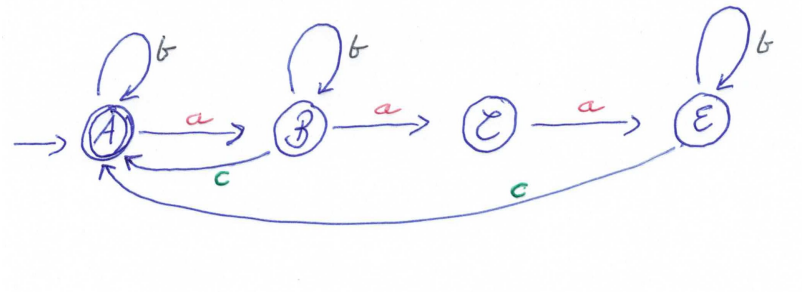


Figure 18: Smaller DFA equivalent to that in Figure 17.

15a (70 p) Consider the following simplified version of the grammar for arithmetic expressions with precedence that we saw in class, where E is the starting symbol:

$$E \to E + E_2 \tag{4a}$$
$$E \to E_2 \tag{4b}$$
$$E_2 \to E_2 * E_3 \tag{4c}$$
$$E_2 \to E_3 \tag{4d}$$
$$E_3 \to \textbf{num} \tag{4e}$$
$$E_3 \to (\, E \,) \tag{4f}$$

Is this an LL(1) grammar?

If your answer is yes, then construct *FIRST*, *FOLLOW*, and *Nullable*, and give pseudo-code for a recursive descent parser.

If your answer is no, then explain why the grammar fails to be LL(1). Is it possible to build a predictive parser for the language in some other way by using more characters of look-ahead?

Solution: This grammar is not LL(1). Suppose that we are starting to parse the input and see a token **num**. Then with just this one token of look-ahead, there is no way of knowing whether we should use $E \to E + E_2$ or $E \to E_2$. This is just another way of saying that the *FIRST*-sets of these two productions are not disjoint.

Unfortunately, adding more characters of look-ahead does not help. With two characters of look-ahead we cannot know whether, e.g., (**num** should be parsed as $(\,E\,)$ or $(\,E\,)+E$, and if we add more characters of look-ahead, then we can also add extra parantheses to this problematic expression.

15b (100 p) Consider a grammar for parenthesized lists of **num** tokens as follows, with S as the starting symbol:

$$S \to P\ S \tag{5a}$$
$$S \to B\ S \tag{5b}$$
$$S \to N \tag{5c}$$
$$P \to (\,S\,) \tag{5d}$$
$$B \to [\,S\,] \tag{5e}$$
$$N \to \textbf{num}\ N \tag{5f}$$
$$N \to \tag{5g}$$

Is this an LL(1) grammar?

If your answer is yes, then construct *FIRST*, *FOLLOW*, and *Nullable*, and give pseudo-code for a recursive descent parser.

If your answer is no, then explain why the grammar fails to be LL(1). Is it possible to build a predictive parser for the language in some other way by using more characters of look-ahead?

Solution: Getting straight to the point, this is an LL(1)-grammar. We show this by constructing *FIRST* and *Nullable* as in Section 2.7 and *FOLLOW* as in Section 2.9 in Mogesen's notes, after having added the new production $S' \to S\ \$$ to the grammar as also suggested in the notes, and then using this information to show that one token of look-ahead is sufficient for correct predictive parsing.

First, it is easy to verify that S and N are *Nullable*, as are the productions $S \to N$ and $N \to$, but that no other nonterminals or productions are *Nullable*. We get the following table for *FIRST* and *Nullable* (which is straightforward enough that we just write it down directly).

Production	*FIRST*	*Nullable*
$S' \to S\ \$$	$(, [, \textbf{num}, \$$	No
$S \to P\ S$	$($	No
$S \to B\ S$	$[$	No
$S \to N$	**num**	Yes
$P \to (\,S\,)$	$($	No
$B \to [\,S\,]$	$[$	No
$N \to \textbf{num}\ N$	**num**	No
$N \to$	$\emptyset$	Yes

Using the *FIRST*-sets we just computed, we can extract the conditions for the *FOLLOW*-sets from the productions as listed below.

Production	Condition
$S' \to S\$$	$\{\$\} \subseteq FOLLOW(S)$
$S \to P\ S$	$\{(, [, \mathbf{num}\} \subseteq FOLLOW(P)$, $FOLLOW(S) \subseteq FOLLOW(P)$
$S \to B\ S$	$\{(, [, \mathbf{num}\} \subseteq FOLLOW(B)$, $FOLLOW(S) \subseteq FOLLOW(B)$
$S \to N$	$FOLLOW(S) \subseteq FOLLOW(N)$
$P \to (\ S\ )$	$\{)\} \subseteq FOLLOW(S)$
$B \to [\ S\ ]$	$\{]\} \subseteq FOLLOW(S)$
$N \to \mathbf{num}\ N$	
$N \to$	

Applying the iterative procedure in Mogesen's notes, we compute the *FOLLOW*-sets for four iterations

Nonterminal	Iteration 1	Iteration 2	Iteration 3	Iteration 4
S	$\emptyset$	$),], \$$	$),], \$$	$),], \$$
P	$\emptyset$	$(, [, \mathbf{num}$	$(, [, \mathbf{num},),], \$$	$(, [, \mathbf{num},),], \$$
B	$\emptyset$	$(, [, \mathbf{num}$	$(, [, \mathbf{num},),], \$$	$(, [, \mathbf{num},),], \$$
N	$\emptyset$	$\emptyset$	$),], \$$	$),], \$$

until we reach the fixpoint.

We have now collected all the information needed to determine whether the grammar under study is an LL(1)-grammar or not. Consulting Mogensen's note, we read that a given grammar is LL(1) precisely when for every nonterminal N it holds that it is correct to choose a production $N \to \alpha$ for the look-ahead c if

- $c \in FIRST(\alpha)$, or
- α is *Nullable* and $c \in FOLLOW(\alpha)$,

without ever having more than one possible production to choose from. (That is, it should never be the case that the condition above applies simultaneously for the same look-ahead token for two different productions $N \to \alpha_1$ and $N \to \alpha_2$.) We need to argue, based on the information that we have gathered so far, that this is true for the grammar we are considering.

If we expand the first table we build above with the *FOLLOW*-sets for the nullable productions, then we get

Production	*FIRST*	*Nullable*	*FOLLOW*
$S' \to S\$$	$(, [, \mathbf{num}, \$$	No	
$S \to P\ S$	$($	No	
$S \to B\ S$	$[$	No	
$S \to N$	$\mathbf{num}$	Yes	$),], \$$
$P \to (\ S\ )$	$($	No	
$B \to [\ S\ ]$	$[$	No	
$N \to \mathbf{num}\ N$	$\mathbf{num}$	No	
$N \to$	$\emptyset$	Yes	$),], \$$

From this table we can read off, among other things, the following interesting pieces of information:

- When starting to parse, if the first token is something other than $(, [, \mathbf{num}$ (or end-of-file), then we have an error immediately.
- When parsing S and deciding on which production to apply, we should choose

1. $S \to P\ S$ if the look-ahead is (;
2. $S \to B\ S$ if the look-ahead is [;
3. $S \to N$ if the look-ahead is one of **num**,),], \$.

- When parsing N, we should choose
  1. $N \to \mathbf{num}\ N$ if the look-ahead is **num**;
  2. $N \to$ if the look-ahead is one of),], \$.
- Regarding P and B, there can be no ambiguity since these nonterminals only have one production each.

In particular, we see that there can never be any conflict regarding which production to choose, but that one token of look-ahead is sufficient. This shows that we are indeed dealing with an LL(1)-grammar, just as claimed.

To turn an LL(1)-grammar into a recursive descent parser, the idea is that every nonterminal will correspond to a method/function/procedure, and the method for N will choose the appropriate production $N \to \alpha$, and then make calls in the right order to the (methods for the) nonterminals occurring in α and also parse any terminals/tokens. For more details, see the examples in Mogesen's notes or the handwritten lecture notes by the lecturer.