CS 577: Introduction to Algorithms Fall 2020

Homework 2 Solutions

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

(a) Design an algorithm to compute the binary representation of 10^n in $O(n^{\log_2 3})$ time.

(b) Design an algorithm that converts a given n-digit decimal number to binary in $O(n^{\log_2 3})$ time.

Part (a)

The idea is to use fast powering (HW 1, prob. 2) with the nontrivial divide and conquer integer multiplication algorithm from class (Integer-Multiplication in the lecture notes). Pseudocode is in algorithm 1.

Algorithm 1

```
Input: $n \in \mathbb{Z}$, $n \geq 1$
Output: $10^n$ in binary
1: procedure Power-Of-Ten($n$)
2:     if $n = 1$ then
3:         return 1010
4:     else
5:         $c \leftarrow$ Power-Of-Ten($\lfloor n/2 \rfloor$)
6:         $s \leftarrow$ Integer-Multiplication($c$, $c$)
7:         if $n$ is even then
8:             return $s$
9:         else ($n$ is odd)
10:            return $(s \cdot 2^3) + (s \cdot 2)$ where multiplication by $2^k$ is implemented by shifting
```

Correctness We argue correctness by induction on n. In the base case, $n = 1$, and the algorithm returns 1010, which is the binary representation of ten. For the inductive step, $n > 1$, and the algorithm enters its recursive case. By the inductive hypothesis, c is the binary representation of $10^{\lfloor n/2 \rfloor}$. By correctness of Integer-Multiplication, s is the binary representation of $10^{2\lfloor n/2 \rfloor}$. When n is even, $10^{2\lfloor n/2 \rfloor} = 10^n$, so the algorithm is correct to return s. When n is odd, the expression $(s \cdot 2^3) + (s \cdot 2)$ computes $s \cdot 10$. Since $s \cdot 10 = 10^{2\lfloor n/2 \rfloor + 1} = 10^n$, the algorithm is correct in this case as well. Correctness for all n now follows by induction.

Running Time We use the recursion tree method. The shape of the tree is a line, with input size n at the root and decreasing by half with each step down the tree. The depth of the tree is at most $\log n$. The work at a node with input size ℓ consists of a call to Integer-Multiplication with $O(\ell)$-bit inputs, plus sums and shifts of numbers of $O(\ell)$ bits. This work is bounded by $c\ell^{\log_2 3}$

for some constant c. At depth d, the input size ℓ is bounded by $n/2^d$. Summing this over all d, the total work of the algorithm is

$$\sum_{d=0}^{\log n} c\left(\frac{n}{2^d}\right)^{\log_2 3} = cn^{\log_2 3}\sum_{d=0}^{\log n}\left(\frac{1}{3}\right)^d \le cn^{\log_2 3}\sum_{d=0}^{\infty}\left(\frac{1}{3}\right)^d = cn^{\log_2 3}\cdot\frac{3}{2},$$

where the final equality uses the formula for summing a geometric series. The overall running time is therefore $O(n^{\log_2 3})$.

Part (b)

The idea is to divide the digits of the input into two halves, recursively compute the binary representations of the number represented by each half, and use the INTEGER-MULTIPLICATION routine and the solution from part (a) to combine the halves into the whole.

In more detail, given an n-digit decimal number X, we divide its digits into two parts of size about $n/2$: $X = L \cdot 10^{\lceil n/2\rceil} + R$, where $L, R < 10^{\lceil n/2\rceil}$ are integers. Next we convert A and B to binary, and use the subroutine from part (a) to compute the binary representation of $10^{\lceil n/2\rceil}$. Finally, we use INTEGER-MULTIPLICATION to compute $L \cdot 10^{\lceil n/2\rceil}$ in binary, and add to this the binary representation of R.

Algorithm 2

Input: X, an n-digit integer given in decimal

Output: the binary representation of X

```
1: procedure DECIMAL-TO-BINARY($X$)
2:     if $n = 1$ then
3:         return binary representation of the digit of $X$ [lookup table omitted for brevity]
4:     else
5:         Let $R$ be the lowest $\lceil n/2\rceil$ digits of $X$, and $L$ the remaining digits.
6:         $L' \leftarrow$ DECIMAL-TO-BINARY($L$)
7:         $R' \leftarrow$ DECIMAL-TO-BINARY($R$)
8:         $P \leftarrow$ POWER-OF-TEN($\lceil n/2\rceil$)
9:         return INTEGER-MULTIPLICATION($L'$, $P$) + $R'$
```

Correctness We argue correctness by induction on n. The n-th assertion is that the algorithm is correct on all inputs X with at most n decimal digits. For the base case, $n = 1$, and the algorithm is correct (provided the omitted lookup table is correct). For the inductive step, $n > 1$. We have $X = L \cdot 10^{\lceil n/2\rceil} + R$. By the inductive hypothesis, L' and R' are the binary representations of L and R respectively. By correctness of part (a), P is the binary representation of $10^{\lceil n/2\rceil}$. Finally, by correctness of INTEGER-MULTIPLICATION, the value returned by the algorithm is $L' \cdot P + R'$, the binary representation of $L \cdot 10^{\lceil n/2\rceil} + R = X$, as desired. Correctness for all n now follows by induction.

Running Time We use the recursion tree method. Each non-leaf node in the recursion tree has two children, and the input size n shrinks by half with each level of recursion, so the tree has the same shape as MERGESORT. At a node with input size ℓ, the non-recursive work done is essentially the calls to Power-Of-Ten and Integer-Multiplication on inputs of size $O(\ell)$. The subroutines each run in $O(\ell^{\log_2 3})$, so for some constant c the work at a node with input size ℓ is $c\ell^{\log_2 3}$.

At depth d, the input size ℓ is at most $\frac{n}{2^d}$. As there are 2^d nodes at depth d, the total work at level d is

$$2^d \cdot c\left(\frac{n}{2^d}\right)^{\log_2 3} = cn^{\log_2 3}\left(\frac{2}{3}\right)^d .$$

As in part (a), the sum over all d behaves as a geometric series with base less than 1. So the total work done is $O\left(n^{\log_2 3}\right)$.

Problem 2

Given an array $A[1, \ldots, n]$ of integers and a positive integer k, we want to find a rearrangement B of A such that the subarrays $B_1 \doteq B[1, \ldots, k], B_2 \doteq B[k+1, \ldots, 2k], \ldots$, satisfy the following property: For every $i < j$, every element of B_i is less than or equal to every element of B_j.

Design an $O(n \log(n/k))$ algorithm for this problem. You can assume that all elements in the array are distinct.

We first consider the case where $n = 2k$. Here the problem becomes to rearrange A so that, when divided into two halves, A_1 and A_2, each element of A_1 is less than or equal to every element of A_2. The running time requirement becomes $O(n \log(n/k)) = O(n)$.

One way to rephrase the problem is that we want to rearrange A so that A_1 consists exclusively of elements at most the median, and A_2 consists of elements at least the median. Using linear-time selection, (Fast-Select in the notes), we can find the median of A. Using the splitting subroutine (Split in the notes), we can use the median to divide A into the requisite parts. Both algorithms are linear-time, so this addresses the $n = 2k$ case.

Plugging the above idea into a typical divide-and-conquer scheme, we can handle the case where n/k is a power of 2. We Fast-Select the median of A, use it to Split A into its lower and upper halves, and then recurse on the left and right halves. This idea works also for general k, but we need to take care to select an element of A on the boundary of the middle A_i instead of just the median of A. Complete pseudocode is below.

Algorithm 3

Input: $A[1, \ldots, n], k$. A is an array of integers, and k is a positive integer at most n.

Output: A permuted copy B of A so that the subarrays $B_1 \doteq B[1, \ldots, k], B_2 \doteq B[k+1, \ldots, 2k], \ldots$ satisfy that for every $i < j$, every element of B_i is less than or equal to every element of B_j.

```
1:  procedure Partial-Sort($A$)
2:      if $n = k$ then
3:          return $A$
4:      else
5:          $b \leftarrow \lceil n/k \rceil$ (the number of subarrays $A_i$ of size $k$)
6:          $r \leftarrow k\lfloor b/2 \rfloor$ (number of elements in $A_1, \ldots, A_{\lfloor b/2 \rfloor}$)
7:          $p \leftarrow$ Fast-Select($A$, $r$)
8:          $(L, R) \leftarrow$ Split($A$,$p$)
9:          Append $p$ to the right of $L$
10:         $L' \leftarrow$ Partial-Sort($L$, $k$)
11:         $R' \leftarrow$ Partial-Sort($R$, $k$)
12:         return $L' \cdot R'$
```

Correctness We argue correctness for all inputs A, k by induction on the length n of A. The n-th assertion is that for all A of length n, Partial-Sort is correct on input A, k. The base case of our induction is when $n = k$. In this case there is no constraint on the ordering of A, so the algorithm returning A unchanged is a correct result.

For the inductive step, we have $n > k$. Let $b = \lceil n/k \rceil$ be the number of subarrays $A_1, A_2, \ldots$, each of size k, and let $r = k\lfloor b/2 \rfloor$ be as in the algorithm. By the specifications of Fast-Select and

SPLIT, after line 9, the smallest r elements of A are in array L, and the largest $n-r$ elements are in in array R. Since $r < n$ and $n - r < n$, the inductive hypothesis implies that the recursive calls to PARTIAL-SORT are correct, and thus both L' and R' obey the output condition. Let $B = L' \cdot R'$, the value returned by the algorithm. We now check that, for every $i < j$, each element of B_i is less than or equal to each element of B_j. There are three cases:

- When i and j are both at most $\lfloor b/2 \rfloor$, this follows from correctness of the first recursive call to PARTIAL-SORT.
- When i and j are both larger than $\lfloor b/2 \rfloor$, this follows from correctness of the second recursive call to PARTIAL-SORT.
- When $i \leq \lfloor b/2 \rfloor$ and $j > \lfloor b/2 \rfloor$, each element of B_i is among the smallest r elements of A, and each element of B_j is among the largest $n-r$ elements of A, so correctness holds in this case as well.

This completes the inductive step. Correctness for all n follows by induction.

Running Time We use the recursion tree method. Each non-leaf node has two children, and the input size n is reduced by half with each level of recursion. The leaves correspond to when $n = k$, so the depth is the number of times n must be halved (up to rounding) before becoming equal to k. This is $O(\log(n/k))$. For any node in the recursion tree, the work local to that node consists of calls to FAST-SELECT and SPLIT, which are linear time. So for some constant c, this work is bounded by $c\ell$. Nodes at depth d have input size $\ell \leq n/2^d$. Summing the work across all 2^d nodes at level d, we get a bound of cn. Summing these across all levels, we get a total running time of $cn \log(n/k) = O(n \log(n/k))$.

Problem 3

In the knapsack problem, you are given a nonnegative weight limit W and arrays $w[1,\ldots,n]$ and $v[1,\ldots,n]$ of nonnegative integers for some $n \geq 1$, where $w[i]$ represents the weight of the ith item and $v[i]$ its value. You want to know the maximum total value that you can achieve by a subset of the items such that the total weight does not exceed W.

Consider the knapsack problem with the following additional restriction: if you include an item with value v then you need to also include every item with value greater than v.

Design an algorithm that solves this problem in $O(n)$ time irrespective of the order in which the items are given. You can assume that all values in the array $v[1,\ldots,n]$ are distinct.

We solve this problem using linear-time selection. The key observation is that if we include the item i with median value $v[i]$, then we also need to include every other item with value greater than i. This means that if the sum of the weights of these items exceeds W, we can forget about including items with lower value, and only recurse on the items with greater value. On the other hand, if these items fit then we must include all of them in the knapsack, and we are left with solving the simpler problem of fitting the remaining items in a smaller knapsack. In any case, we are able to halve the input size, while only performing a linear amount of work per recursive call.

In order to present the pseudocode for the algorithm, we consider a slight variation of the procedure Split in the lecture notes that receives an additional array as input. Whenever Split makes a swap in the original array, Split' makes the swap on both arrays. Additionally, Split' puts the pivot element in the left array. Complete pseudocode is below in Algorithm 4, we include the code for Split' for completeness.

Correctness We argue correctness for all inputs $W, w[1,\ldots,n], v[1,\ldots,n]$ by induction on n. The n-th assertion states that the algorithm Knapsack correctly solves the given problem for n items.

The base case is when $n = 1$. In this case, we have a single item with weight $w[1]$ and value $v[1]$. The maximum value then is either 0, if $w[1] > W$, or $v[1]$, if $w \geq W$. The code in lines 2-6 handles this case in the algorithm.

For the inductive step, we consider $n > 1$. The inductive hypothesis states that any calls made to Knapsack with arrays of length less than n are correct.

Let V be the maximum total value that can be obtained while respecting both the weight limit and the additional constraint for the problem. Let m be the item with median value v_m. Also, let W' and V' be the sum of weights and values, respectively, of all items with value greater than v_m. There are two possibilities in relation to W':

- If $W' > W$, then we cannot add item m to the knapsack. Also, because of the additional constraint, we cannot add any item with value less than v_m in the knapsack. With this in mind, the maximum V can be obtained by solving the same problem, with weight limit W and only considering items with value greater than v_m. By the inductive hypothesis, the call to Knapsack performed by the algorithm in line 13 computes this value correctly.

- If $W' \leq W$, then it must be the case that every item with value greater than v_m is included in the knapsack, and thus $V \geq V'$. Moreover, the maximum value is attained when we maximize the value we can get out of the remaining items with value less than or equal to v_m. In this case, V equals V' plus the maximum value we can obtain using only items with value v_m or less (this includes item m) considering a backpack with weight $W - W'$. By the inductive hypothesis, the call to KNAPSACK performed by the algorithm in line 15 computes this value correctly. We conclude that the algorithm is correct.

Running Time. We use the recursion tree method, as in Figure 1. Each non-leaf node has a single child, and the input size n is reduced by half with each level of recursion. The leaves correspond to when $n = 1$, so the depth of the tree is $\log n$. For any node in the recursion tree, the work local to that node consists of calls to FAST-SELECT and SPLIT', each of which take linear time, and computing two sums of $\lfloor n/2 \rfloor$ elements, which also take linear time. Thus, the local amount of work performed per node is bounded by $c\ell$, where ℓ is the input size for that node. At depth d, nodes have input size $n/2^d$. Figure 1 is a graphical representation of this recursion tree.

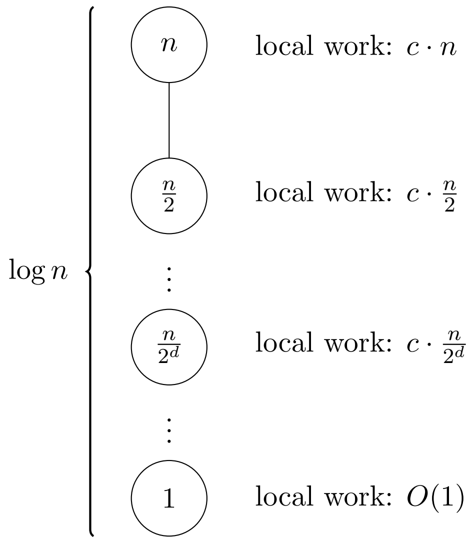


Figure 1: Recursion tree for the KNAPSACK algorithm.

With this in mind, the total work performed by the algorithm can be bounded by

$$\sum_{d=0}^{\log n} \frac{cn}{2^d} = cn \sum_{d=0}^{\log n} \frac{1}{2^d} \leq cn \cdot 2 = O(n).$$

Algorithm 4

```
Input: $W, w[1,\dots,n], v[1,\dots,n]$. $W$ is a nonnegative integer, $w$ and $v$ are arrays of nonnegative
    integers. All values in $v$ are distinct.
Output: $V$, the maximum value obtained by selecting a subset of items in $[1,\dots,n]$ such that the
    sum of weights of these items does not exceed $W$ and whenever an item is included, every other
    item with higher value is also included.
procedure KNAPSACK($W, w, v$)
    if $n = 1$ then
        if $w[1] \leq W$ then
            return $v[1]$
        else
            return 0
    else
        $v_m \leftarrow$ FAST-SELECT($v[1,\dots,n]$, $\lceil n/2 \rceil$)
        $(v_{\text{left}}, v_{\text{right}}, w_{\text{left}}, w_{\text{right}}) \leftarrow$ SPLIT'($v[1,\dots,n], v_m, w[1,\dots,n]$)
        $W' \leftarrow$ sum of weights in $w_{\text{right}}$ (total weight of items with value $>$ the median value)
        $V' \leftarrow$ sum of weights in $v_{\text{right}}$ (total value of items with value $>$ the median value)
        if $W' > W$ then
            return KNAPSACK($W$, $w_{\text{right}}$, $v_{\text{right}}$)
        else
            return $V'$ + KNAPSACK($W - W'$, $w_{\text{left}}$, $v_{\text{left}}$)
```

```
Input: Two arrays of integers $A[1,\dots,n]$, $B[1,\dots,n]$, an integer $p$.
Output: Arrays $A_{\text{left}}, A_{\text{right}}, B_{\text{left}}, B_{\text{right}}$ such that $A_{\text{left}}$ contains all elements in $A$ that are less
    than or equal to $p$ and $A_{\text{right}}$ contains all elements that are greater than $p$. Arrays $B_{\text{left}}$ and
    $B_{\text{right}}$ contain elements of $B$ that have consistent indexes with $B_{\text{left}}$ and $B_{\text{right}}$, respectively, i.e.
    if the index of $A[i]$ is $j$ in $A_{\text{left}}$, then the index of $B[i]$ is $j$ in $B_{\text{left}}$.
procedure SPLIT'($A$, $B$, $p$)
    $(A, B) \leftarrow$ empty lists
    for $i = 1$ to $n$ do
        if $A[i] \leq p$ then
            Append $A[i]$ to $A_{\text{left}}$
            Append $B[i]$ to $B_{\text{left}}$
        else if $A[i] > p$ then
            Append $A[i]$ to $A_{\text{right}}$
            Append $B[i]$ to $B_{\text{right}}$
    return $(A_{\text{left}}, A_{\text{right}}, B_{\text{left}, B_{\text{right}}})$
```

Problem 4

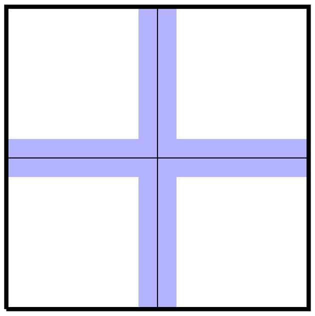
You are given an $n \times n$ grid, and a procedure $V(i,j)$ that assigns an integer value to each position (i,j) in the grid, where i and j are integers such that $1 \le i,j \le n$. Your goal is to find a local minimum (or sink) in the grid (i.e., integers i^*, j^* with $1 \le i^*, j^* \le n$ such that for all neighbors (i,j) of (i^*, j^*) in the grid, $V(i^*, j^*) \le V(i,j)$). The neighbors of (i,j) are $(i-1,j)$, $(i+1,j)$, $(i,j+1)$, $(i,j-1)$; the elements along the diagonals do not count as neighbors.

Design an algorithm that makes $O(n)$ calls to V. Note that the grid has n^2 nodes.

We first observe that any $n \times n$ grid contains a local minimum, since any path constructed by repeatedly stepping toward a neighbor with a strictly smaller label must terminate at a local minimum. While this is indeed an algorithm, it isn't efficient enough as it could potentially take $\Theta(n^2)$ probes.

We use divide-and-conquer to find a local-minimum in $O(n)$ probes. We think of dividing the input grid into four sub-grids, as in Figure 2. Generically speaking, a local minimum of any one of the sub-grids is a local minimum of the overall grid, so the idea is to do some $O(n)$ work to decide which of the four sub-grids to recurse into, and do that. Since n shrinks by half with each layer of recursion, the overall running time will behave like a geometric series, and be $O(n)$ overall.

Figure 2: Dividing a grid into four sub-grids.



This intuition about minima of sub-grids is not quite right. Our first hint of this is that, if every local minimum of a sub-grid were a local minimum of the overall grid, then we could pick *any* of the four sub-grids to recurse into. Thus we would be able to find a local minimum with zero probes! Clearly this is wrong, so something is missing.

And indeed, it is possible for an element to be a local minimum in a sub-grid and not be a local minimum in the overall grid. For example, there may be a smaller neighbor in a different sub-grid. That being said, it *is* true that if every neighbor of a sub-grid's local minimum lies within the same sub-grid, then it is a local minimum of the overall grid. This suggests a more refined strategy: prior to recursing into a sub-grid, we try to ensure that the local minimum found is not on the boundary.

To manage this, first observe that there are only $O(n)$ many positions that have a neighbor in a different sub-grid. We will refer to these positions as boundary positions; they are highlighted as the blue region in Figure 2. We can afford to examine them all. We can use that information to decide which of the four sub-grids to recurse into.

An interesting boundary position is the one that has the smallest label among all the boundary positions. Call this position m. Its label is clearly no larger than the label of any of its neighbors that are also boundary positions. So the only way m is *not* a local minimum of the grid is if it has a smaller neighbor in the *same* sub-grid. We can imagine following a path starting from m and repeatedly stepping to an adjacent position with strictly smaller label, until reaching a local minimum. Because m is the minimum of *all* boundary positions, this path can never return to the boundary of the subgrid. In particular, there has to be a local minimum in the sub-grid containing m that is not on the boundary of that sub-grid. Thus suggests we should recurse into the sub-grid containing m.

This almost works! There is one issue though. The specification of our algorithm is that it finds *some* local minimum. When we recurse onto the sub-grid containing m, there is no guarantee the local minimum found is the one arrived at by repeatedly stepping to a neighbor with strictly smaller label as above. Indeed, the recursive call could very well return a boundary position. We need to strengthen our specification so that it not only returns a local minimum, but one that we can argue is not on the boundary.

To do this, we will require that the local minimum we find in a sub-grid has label no larger than the label of m. The local minimum arrived at by repeatedly stepping to a neighbor with strictly smaller label satisfies this property, so certainly it exists in the sub-grid containing m. Let's carefully write down a specification for our algorithm:

Input: A grid G, a labeling procedure V, and a position t in the grid.

Output: A local minimum of the grid that has label no larger than the label of t.

At the top layer of the recursion there is no need for the additional argument; we can pass an arbitrary position. Since we adjusted the specification, we need to ensure our algorithm matches it as well. In particular, we now have the additional input of some position t in the grid, and need to make sure the local minimum we find has label smaller than the label of that position. If the label of m is at most the label of t, we are fine. Otherwise, the label of t is smaller than the label of m, and thus also smaller than every label of a boundary position, so we can instead recurse into the sub-grid containing t. This gives the following algorithm:

Correctness: We prove correctness by induction on n, the size of the grid. The n-th assertion is that FindGridLocalMinimum matches its specification for all grids of size at most n.

Base Case: For the case in which the grid has size $n = 1$, there is only one position, namely t. As it has no neighbors, it is a local minimum, so the algorithm is correct.

Inductive step: When $n > 1$, the algorithm falls into its recursive case. Let m be as in the algorithm and B represent the set of boundary positions. In both the case $V(t) \geq V(m)$ and the case $V(t) < V(m)$, we return the result of a recursive call FindGridLocalMinimum(G', u), where G' is a subgrid of G and u is a node in G' such that $V(u) \leq V(b)$ for every $b \in B$ and $V(u) \leq V(t)$. By the inductive hypothesis, the call returns a local minimum m' of the subgrid G' with $V(m') \leq V(u)$. Because m' is a local minimum in G', every neighbor of m' in G' has no smaller label. Every other neighbor of m' is a position $b \in B$. Since $V(m') \leq V(u) \leq V(b)$, it follows that every neighbor of m' has no smaller label, and so m' is a local minimum in G. Finally, $V(m') \leq V(u) \leq V(t)$, so the condition that $V(m') \leq V(t)$ is satisfied as well.

Algorithm 5 FindGridLocalMinimum

Input: An $n \times n$ grid G; a position t in G; access to a procedure V assigning an integer to each position in G.

Output: A local minimum of G with label at most $V(t)$

```
procedure FindGridLocalMinimum($G, t$)
    if $n = 1$ then
        return $t$
    else
        Divide $G$ as in Figure 2
        Let $B$ represent the boundary positions
        Let $m$ be a position s.t. $V(m) = \min_{b \in B} V(b)$
        if $V(t) \geq V(m)$ then
            Let $G'$ be the quadrant containing $m$
            return FindGridLocalMinimum($G', m$)
        else
            Let $G'$ be the quadrant containing $t$
            return FindGridLocalMinimum($G', t$)
```

Analysis: We use the recursion tree method. The shape of the recursion tree is a line, with input size n at the root and decreasing by half with each layer of recursion. The work done at a node with input size ℓ is some constant amount of book-keeping plus the cost of finding the minimum boundary position. As there are at most 4ℓ boundary positions, and finding a minimum takes linear time, this cost is $c\ell$ for some constant c. Nodes at depth d of the recursion tree have input size $\ell \leq n/2^d$; summing the work per node over the whole tree gives the bound $cn + c\frac{n}{2} + \cdots \leq 2cn = O(n)$ on the total work done.

Problem 5

You are given a topographical map that provides the maximum altitude along the direct road between any two neighboring cities, and two cities s and t. Design a linear-time algorithm that finds a route from s to t that minimizes the maximum altitude. All roads can be traversed in both directions.

For this problem, we construct from our map a graph G on n vertices, with each vertex representing a city. Two vertices are connected by an edge if there is a direct road between them on our map, and the weight of each edge is the maximum altitude that occurs on that road. In this view, we are looking for a path from s to t such that the largest weight on that path is as small as possible.

To achieve this, we try to construct a path from s to t using the lightest edges possible. As a first step, we find the smallest threshold τ such that we can get from s to t using only edges with weight at most τ. Once we find τ, we can use it to find a path in which all edges have weight at most τ. First we discard all edges with weight larger than τ. Then we perform a breadth first search on the remaining graph to find a path. Deleting the edges and performing a breadth first search can be done in $O(n+m)$ time. So from here on out, we focus on finding the smallest threshold τ such that we can get from s to t using only edges with weight at most τ.

We start by doing a breadth-first search on G to ensure that there is, in fact, a path from s to t. This takes time $O(n+m)$. If there is no path, then we are done. Otherwise, note that we can focus just on the connected component of G that contains s and t; the rest of the graph is irrelevant to us. From now on, we will assume G is connected. In this case, $m \geq n-1$, and we can express the running time of BFS and related algorithms in terms of m only. (This helps to simplify the analysis.) We also know that there exists a path from s to t. The question then becomes, how small can we set a threshold τ so that there exists a path from s to t among edges with weight at most τ?

For this we can use divide and conquer, specifically binary search. For a given threshold τ, we can check whether there is a path from s to t that uses only edges of weight at most τ. This amounts to running a breadth-first search from s that ignores edges of weight more than τ. If the path exists, we need not consider larger values of τ; if no such path exists, we need not consider smaller values. The question remains how to pick the values of τ to test.

It is clear that the value of τ we ultimately want is equal to the weight of some edge in the graph. Given that, we want to search for τ among the list of edge weights. Following the pattern of binary search, we maintain a list of viable thresholds (initially all edge weights), and use the median of that list as τ in the above test. Based on the result of the test, we either recursively search among thresholds less than the median or larger than the median; either way removes half the candidates.

That gives a complete algorithm. Initially there are m viable thresholds, so the binary search requires $\log m$ iterations to complete. Each iteration takes time $O(m)$ to run, as the test has to run a BFS on the whole graph, including the edges with nonviable weights. This gives a running time of $O(m \log m)$. This is too slow! We need to make it faster.

To do better, we need to keep the BFS from considering edges with nonviable weights. If we can do this, the work done by the BFS will shrink with the list of viable edge weights, i.e., it will halve from one step to the next. In total, the overall cost of the binary search behaves like a geometric series, and the running time becomes $O(m)$.

Edges with weights too large to be viable are easy to handle. We just delete them from the graph. Then the BFS does not have to waste time seeing and ignoring them.

As for edges with weights too small to be viable, we want the BFS to treat sets of vertices connected by such edges in one fell swoop. A way to do this is just to merge all vertices that are connected by such edges. In a bit more detail, we first form a subgraph G_1 of G with the same vertices and with only the edges of G with weights too small to be viable. From G_1, we form another graph, G_2, by merging the vertices in each connected component of G_1. G_2 has one vertex per connected component of G_1, and an edge for each each edge in G that is still viable as the minimum maximum altitude. Then we recurse on G_2. It only has edges corresponding to viable edges of G, so it is indeed half the size of G. We can compute G_2 from G in linear time: forming G_1 takes linear time; the connected components of G_1 are found in linear time with a breadth-first search; and figuring out what endpoints each edge should have in G_2 takes only a single pass through the viable edges of G.

This gives us our final procedure: we find the median edge-weight in linear time, and decide whether we can get from s to t using only the lighter edges. If we can, we delete all the heavier edges and recurse on a single subproblem half the size of the original. If we cannot, we merge vertices when they are connected by light edges, after which only heavy edges remain, and we again have a single subproblem of half the size. Pseudocode is given in Algorithm 6 on page 14. It is slightly more complex than this summary, as it deals with the possibility of repeated edge weights.

Correctness Correctness essentially follows from the above discussion; we give a more formal argument here. Correctness of MinWeightPath follows directly from correctness of MinWeight-PathConnected, so we will just establish the latter.

To prove correctness of MinWeightPathConnected, we do induction on m. The m-th assertion is that the algorithm is correct for all graphs G with at most m edges. In the base case, there is one edge. Since s and t are distinct, but connected, they are connected by the unique remaining edge. Therefore, the algorithm correctly returns the weight of that edge.

For the inductive step, $m > 1$. Let τ be as in the algorithm. Every path connecting s and t by edges with weight less than τ is a path in G_1, and every path connecting s and t by edges with weight at most τ is a path in $\widehat{G}_1$. We now break into three cases:

Case 1. There exists a path connecting s and t for which every edge has weight less than τ. In this case, the path continues to exist in G_1, and the algorithm goes to line 18, where it constructs and recurses on G_1'. By construction, G_1' is connected. Moreover, every path from s to t with maximum edge weight less than τ remains in G_1'. Finally, at least one edge is removed from G to form G_1 (hence also G_1'), so we can apply the inductive hypothesis to conclude the recursive call is correct. This implies correctness in Case 1.

Case 2. There exists a path connecting s and t with maximum weight exactly τ, but there are no paths with smaller maximum weight. In this case, s and t are not connected in G_1, but are connected in $\widehat{G}_1$, and the algorithm correctly returns τ on line 21.

Case 3. All paths connecting s and t have maximum-weight larger than τ. In this case, s and t are disconnected in both G_1 and $\widehat{G}_1$, so the algorithm goes to line 23, where it constructs G_2 and recurses to find a max-weight path from c_s to c_t. Since s and t are not connected in $\widehat{G}_1$, c_s and c_t are distinct vertices of G_2. Every path in G gives rise to a similar path in G_2 by following the edges with weight larger than τ. In particular, G_2 is connected, and the

Algorithm 6

Input: Weighted graph G with m edges, distinct vertices s, t

Output: Minimum τ such that there is a path from s to t using only edges with weight at most τ, or indicate that no such path exists.

```
 1: procedure MinWeightPath($G, s, t$)
 2:     Compute connected components of $G$
 3:     if $s = t$ then
 4:         return no need to walk anywhere (max altitude is undefined)
 5:     else if $s$ and $t$ are not connected then
 6:         return impossible to walk from $s$ to $t$
 7:     else
 8:         $G' \leftarrow$ connected component containing $s$ and $t$
 9:         return MinWeightPathConnected($G', s, t$)
```

Input: Connected, weighted graph G with m edges, distinct vertices s, t

Output: Minimum τ such that there is a path from s to t using only edges with weight at most τ

```
10: procedure MinWeightPathConnected($G, s, t$)
11:     if $m = 1$ then
12:         return the weight of the edge in $G$
13:     else
14:         $W \leftarrow$ list of all weights of edges in $G$
15:         $\tau \leftarrow$ Fast-Select($W, \lfloor m/2 \rfloor$)
16:         $G_1 \leftarrow$ copy of $G$ with all edges of weight $\geq \tau$ removed
17:         $\widehat{G}_1 \leftarrow$ copy of $G$ with all edges of weight $> \tau$ removed
18:         Compute connected components of $G_1$ and $\widehat{G}_1$
19:         if $s$ and $t$ are connected in $G_1$ then
20:             $G'_1 \leftarrow$ connected component of $G_1$ containing $s$ and $t$
21:             return MinWeightPathConnected($G'_1, s, t$)
22:         else if $s$ and $t$ are connected in $\widehat{G}_1$ then
23:             return $\tau$
24:         else
25:             $G_2 \leftarrow$ new graph with one vertex per connected component of $\widehat{G}_1$
26:             for each edge $\{u, v\}$ in $G$ with weight $w > \tau$ do
27:                 Add an edge to $G_2$ between the components of $u$ and $v$ with weight $w$
28:             $c_s, c_t \leftarrow$ components of $s, t$, respectively
29:             return MinWeightPathConnected($G_2, c_s, c_t$)
```

smallest maximum weight of a path from s to t in G_2 is the same as the smallest maximum weight of a path in G. Moreover, $\widehat{G}_1$ contains at least one edge, so G_2 has fewer edges than G. Therefore the induction hypothesis applies to the recursive call on line 27, and this implies the algorithm is correct.

As this handles all the cases, the induction step is complete, and correctness for all inputs follows by induction.

Running time analysis We start with MinWeightPathConnected. We use the recursion tree method. The recursion tree is shaped as a line. We measure the input size by the number of edges in G. At the root, this is m. From one node to the next, the number of edges shrinks by at least half. The work local to a node consists of a linear-time selection routine, a few breadth-first searches, and some elementary linear-time operations. Thus for some constant c, each node of the recursion tree working on ℓ edges does at most $c\ell$ work. Adding up all this work, we get

$$cm + c\frac{m}{2} + \cdots + \cdots c\frac{m}{2^{\log m}} \leq cm \cdot \left(1 + \frac{1}{2} + \cdots\right) = 2cm = O(m)$$

As for MinWeightPath, computing connected components takes $O(n+m)$ time, and the call to MinWeightPathConnected takes $O(m)$ time. Combined, this is $O(n+m)$ time.