

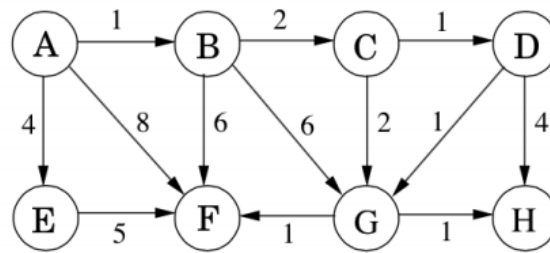
CMPS 102: Homework #4

Due on Tuesday, April 28st, 2015

John Allard 1437547

Problem 1

Suppose Dijkstra's algorithm is run on the following graph, starting at node A.

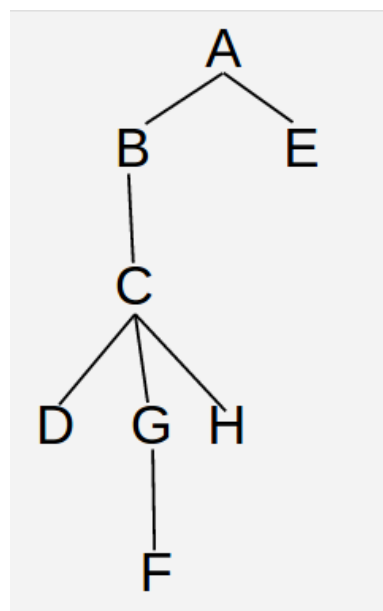


1. Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

The table below shows each iteration as a row, with later iterations being below earlier iterations. The numbers in each table section represent the shortest distance known from the source (vertex A) to the vertex corresponding to that column during the iteration given by the row number. The distances to A are always zero because A is the source. For brevity, I use a '.' instead of an ∞ to represent infinite distances.

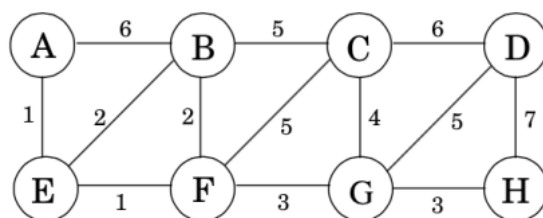
d(x)	a	b	c	d	e	f	g	h
	0
	0	1	.	.	4	8	.	.
	0	1	3	.	4	7	7	.
	0	1	3	4	4	7	5	4
	0	1	3	4	4	7	5	4
	0	1	3	4	4	7	5	4
	0	1	3	4	4	6	5	4
	0	1	3	4	4	6	5	4

2. Show the final shortest-path tree.



Problem 2

Consider the following graph



1. What is the cost of its minimum spanning tree?

The total cost of the minimum spanning tree is 14, it consists of the following edges :

$$[(A, E, 1), (E, B, 2), (E, F, 1), (F, G, 3), (G, C, 4), (G, H, 3)]$$

2. How many minimum spanning trees does it have?

By Cayley's theorem, there are n^{n-2} spanning trees, given that $n = 8$, the number of spanning trees is $8^6 = 262144$.

3. Suppose Kruskal's algorithm is run on this graph. In what order are the edges added to the MST? For each edge in this sequence give a cut that justifies its addition.

Below is a table that answers the question above. It shows the steps of the algorithm going down the table, with the groups of vertices being shown in the other columns. Notice they all start out as singletons before being formed into a single tree.

iteration	A	B	C	D	E	F	G	H
1	AE	B	C	D		F	G	H
2	AEF	B	C	D			G	H
3	AEFB		C	D			G	H
4	AEFBG		C	D				H
5	AEFBGH		C	D				
6	AEFBGHC			D				
7	AEFBGHCD							

Problem 3

You are given an unbounded array $A[1], A[2], A[3], \dots$ containing distinct integers sorted in ascending order. Describe an efficient algorithm that takes an integer k as input and finds out whether k is in the array in time $O(\log p)$ time where p is the number of integers in the array that are strictly less than k .

Since the required runtime is $O(\log(p))$, I know I am going to have to shrink the search space by a constant factor for each iteration. This will result in an exponentially-conquering search of the space, which will reduce the run-time needed to process n elements from linear to logarithmic. To do this, I will use a technique inspired by the exponential backoff algorithm. Instead of looking at each element in the array starting from the first element and continuing until we find the one we're looking for, I will look at every 2^n th item for $n \in \mathbb{N}$. Once I find an element larger than the desired element, I can simply perform a binary search in

between the last two elements I have looked at. Since the entire array is ordered, binary search will work correctly and the element will be found. The algorithm is given below and they are discussed in detail below that. :

```
// A = unbounded, sorted array on integers starting at index 1
// x is the element we are searching A for
1.  findx(A, x) :
2.      ind = 1
3.      if x = A[ind] : // special case, first item in array
4.          return true
5.      else if x < A[ind] : // if its less than element #1 it can't exist
6.          return false
7.      while A[ind] <= x : // while we haven't passed p
8.          ind = ind*2 // double the index value
9.      result = BinarySearch(A, x, ind/2, ind)
10.     if result == -1 : // wasn't found in the sub-array
11.         return false
12.     else :
13.         return true // was found in the sub-array

// A = sorted array, x is the element we are searching A for
// l is left index of subarray to search, r is right index
// returns index of x if found, -1 otherwise
1.  BinarySearch(A, x, l, r) :
2.      if r == l :
3.          if x == A[r] :
4.              return r
5.          else :
6.              return -1
7.      ind = (r+l)/2
8.      if x == A[ind] : // we found it
9.          return ind
10.     else if x < A[ind] : // while we haven't passed p
11.         return BinarySearch(A, x, l, ind)
12.     else if x > A[ind] : // while we haven't passed p
13.         return BinarySearch(A, x, ind, r)
```

I've broken up my implementation into two main parts. The first part scans exponentially from the beginning of the array until it finds an element bigger than x , the element we are looking for. At this point, it knows that if the element does exist, it must exist in between the current value which is bigger than x and the last value it looked at which was smaller than x . Because the algorithm now knows which two indices the element must exist between (if it exists at all), we can defer the rest of the work to a simple binary search over the sub-array. If the binary search returns an index value, we now know the element exists and so we return true. If BinarySearch returns -1, then we know it wasn't in the subarray and thus doesn't exist at all in the full array. Proofs of correctness and run-time are given below.

Proof of Correctness :

The correctness of this algorithm relies on the fact that the elements in the array are sorted in ascending order. Normally, given a sorted array, we would just perform a binary search to locate an element. That of

course does not work for this problem because the array is unbounded, but we can reduce this problem to one of binary search with some trickery. The trick is, if we can find two elements, one of which is greater than the item we are looking for and the other of which is smaller, we can reduce this problem to a problem of performing a binary search over the subarray between those two elements. This is enumerated below.

Proof : The following properties are used in the proof.

1. Let's say we are looking for an element x in a sorted array A . If we examine an element of A and notice it is smaller than x , then x must either be further along in the array or it must not exist in the array at all. This is obvious from the properties of a sorted array.
2. Likewise, if we're looking for an element x in a sorted array A , and we examine an arbitrary element of A and notice it is bigger than x , this means that x must either be earlier in the array or not exist at all.

This algorithm is called on an array A and with a value x that we are looking for. It starts by examining the first element in A , if it is x then we go ahead and return true. If the element is greater than x , we know that x can't exist in the array so we go ahead and return false. Those are the two special cases of this algorithm, the general case of the algorithm is handled next. We start by doubling the index, from 1 to 2. If the $A[2]$ element is greater than x , we exit the loop. If it is less than x , we double the index and repeat the loop. This process of querying and doubling the query index repeats until we find some element greater than x , which because this is an unbounded increasing array and x is finite must eventually happen. When this does happen we exit the loop.

When the loop is exited, we have the index of the first element we have come across that is bigger than x . By (2) above, this means that x must either occur earlier in the array or it doesn't exist at all. Likewise, because in the previous iteration we examined the element $A[ind/2]$ and found that element to be less than x , by (1) above we know that x must exist at an index after $ind/2$ or it doesn't exist at all. Combining these two facts together means that x must either be between the indices $ind/2$ and ind in A , or it doesn't exist at all. Thus we know we have reduced the search space of A to a finite number of elements that, if x exists, it must be apart of. This sets us up perfectly for binary search on the subarray $A[ind/2, ind]$, which is called on line 9 of the *findx* function given above. This binary search now looks for the element x , returning its index if it is found and -1 otherwise. If it returns -1, we return false because this means x doesn't exist in the subarray and thus does not exist in A at all, or it is found and its index is returned, in which case we return true. The correctness of this algorithm depends on the correctness of binary search, which I'm assuming I don't have to prove. ///

Proof of Run-time Complexity :

The algorithm is required to run in $O(\log(p))$, where p is the number of elements in A that are strictly less than the item we are searching for. Because this algorithm consists of two components which are executed in sequence, I will prove that their individual run-times are individually $O(\log(p))$, which means that when run back-to-back their total run-time will also be $O(\log(p))$. The first algorithm to be proved is the one that finds an element greater than the one we are looking for by checking an exponentially increasing index value in the array. For this I will use a direct proof.

Let x be an element you are searching for in a sorted, unbounded array A . Let p denote the number of elements in A with values strictly less than x . If x exists in A , it must be at index $p+1$, given the definition of p . This means our algorithm needs to query an element at an index greater than or equal to $p+1$ in order to find an element that is greater than x . The algorithm starts searching at index 1, and doubles the index value before each subsequent query into the array. The number of operations to find an element larger than x is given below :

$$2^k = p + 1$$

$$k = \lg(p + 1)$$

$$k = \lg(p + 1) \leq \lg(p) + 1$$

$$k \leq \lg(p) + 1$$

$$k = O(\log(p))$$

Thus k , the number of array queries needed to get to index $p + 1$ which contains an element greater than x is $O(\log(p))$, as required for the proof.

After we have found an element which is greater than x , we perform a binary search over a sub-array of A . Given that the element larger than x is at index j , we now call binary search on the subarray $A[j/2, j]$, inclusive of the end elements. Since we doubled our search index every iteration, and we stopped on the first element greater than x that we found, we know that j is no larger than $2p$, otherwise we would have found an element larger than x during the last iteration. We also know that $j/2 \geq 1$, since all index values we use are positive. This means that BinarySearch will be called on a subarray of size at-most $2p - 1$. BinarySearch is logarithmic in the number of elements, so the run-time is $O(\log(2p - 1))$, which is of course $O(\log(p))$.

Because both the algorithm to find an element larger than x and the BinarySearch algorithm to actually find x in a sub-array both run in time $O(\log(p))$, the two run in sequence will run in time $O(\log(p)) + O(\log(p))$, which is of course just $O(\log(p))$. ///

Problem 4

We define an array A of n objects has a *dominant* object if at least $\lfloor n/2 \rfloor + 1$ entries of A are identical. Our goal is to design an efficient algorithm to tell whether the array has a dominant object, and, if so, to find that object. Our only access to A is by making a *query* asking whether $A[i] = A[j]$ for any two $i, j \in \{1, 2, \dots, n\}$.

- (a) Design an algorithm to solve this problem with $O(n \log n)$ queries. (**Hint:** Split the array A into two arrays of half the size.)
- (b) Design an algorithm to solve this problem with $O(n)$ queries. (**Hint:** Don't Split. Pair up the elements arbitrarily and get rid of as many as you can, repeatedly.)

Problem 5

KT, problem 17, p 197.)

Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n . You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by $(start-time, end-time)$ pairs.

(6 PM, 6 AM), (9 PM, 4 AM), (3 AM, 2 PM), (1 PM, 7 PM)

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

Problem 6

(KT, problem 4, p 190)

Sorry too long to retype. Use a greedy algorithm. Reason your time bound as well as correctness.

Problem 7

EC: We discussed an algorithm in class that finds the k -th largest element in an array for $n \geq k$ elements in worst case time $O(n)$. This algorithm starts by finding the medians of groups of 5 elements and then finds the medians of the roughly $n/5$ medians.

Why is this algorithm based on groups of size 5? Does it also work with groups of size 3? Why or why not?