**Time Complexity**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

(a):

(b):

(c):

(d):

(e):

**Fibonacci 1**

Prove: .

By trial and error: It appears that for all

To prove: For all positive integers

By induction on . Base case:

Step, assume: Indeed, true that for all

To prove:

LHS:

Suffices to prove:

(by dividing the above by )

It is indeed true that

**Multiplication**

Graphical user interface, text, application

Description automatically generated

Suppose instead of both and being n-bit, is n-bit and is m-bit. What is the worst-case time efficiency of ?

Proposed:

Time Efficiency:

So, final answer:

**Fibonacci 2**

Let be the nth Fibonacci number, Prove .

* Somewhere, we have shown:
* But here, seek to show: There exists positive real , for all in
* Natural proof strategy for “there exists” – construction (i.e., propose some concrete , and show that it works)
* Try some small values for , and see what would work
* Appears that works. Adopt it and check if proof goes through. Now, proof by induction with
* Base case,
* Step: Seek to show given that for all
* by induction assumption

**Fibonacci 3**

Let be the nth Fibonacci number, Prove .

* Recall from logic: not (there exists an egg-laying mammal) for all mammals , is not egg-laying
* Here, : There exists positive real , for all natural ,
* So here, need to prove: Given any positive real , it is true that there exists such that
* By contradiction: Suppose that there exists positive real , such that, for all natural ,
* Then:
* This is true only if is “large” compared to
* What is large? We need
* Try :
* Try :
* Try :
* Try :
* Try :
* Try :
* Prove by induction: for all natural
* Base case : See above
* Step:
* So far: We have shown that indeed, for ,

**Selection Sort**

What is a meaningful characterization of the time efficiency of ?

* Suppose we invoke . In . Suppose now, min is at index in . This index of a min in is at index
* Suppose on input: . Then A evolves in as follows:
* For time efficiency: Need to make meaningful assumption(s)
* Customary Assumptions: (1) is unbounded, (ii) each is bounded
* What should we count? Suppose we all agree that counting # swaps is a meaningful measure for time efficiency
* Then:
* Now, let’s say we want to get a bit more fine-grained. Incorporate (worst case) time for each swap # swaps
* So now, time efficiency:

**Modular Simplification**

1. Is ?

So:

So:

Now:

So:

So:

1. Is divisible by ?

Trick: Keep exponentiating until numbers start to repeat.

Suppose we repeatedly exponentiate :

So: . And . So

Now check whether is divisible by . Indeed:

Repeat with . Repeated exponentiation of :

So:

Now: .

.

1. Is a multiple of ?

is prime. And .

Compare with :

.

**Proving Multiplicative Inverse**

Show that if has a multiplicative inverse modulo , then this inverse is unique (modulo ).

Let’s assume .

Suppose are both multiplicative inverses of . Then:

: Substitution Rule:

Then:

(2): Commutativity

Suppose . Show that is an integer.

So: , which is divisible by .

We say that is a square root of modulo a prime if . Show that if and has a square root modulo , then is such a square root.

Let be the square root of . Then: .

Write . Then,

Keep in mind: .

Try plugging in in the last expression:

Is ?

So, we’re asking: Is ?

So at least one of: or must be .

We know: There exists such that .

We seek to prove: . Sufficient condition for that to be true:

is okay, because is invertible modulo

**Proving Recurrence 1**

Suppose . Prove recurrence correctness.

Case Analysis:

1. If , then . So, the recurrence is correct for the case where
2. If : then . So
3. If : then . So now:

**Proving Recurrence 2**

Let be the quotient and remainder of and be the quotient and remainder of . Prove recurrence correctness.

To be absolutely clear, what are the quotient and remainder of ?

We call the quotient, and the remainder if and only if and are non-negative integers that satisfy:

Proof by case analysis:

1. If , then . So, recurrence is correct for this case.
2. If is even and : then . So:

Where we infer the last line from the facts that: equation is of the form from definition for quotient and remainder, , and we are given .

1. If is odd and :
2. is even, : . So:

This is of the form of the definition of quotient and remainder, except that we need to confirm that indeed lies between and . Which it does not necessarily. Actually, we are given that and therefore not between and . Now we observe:

Now only question that remains: is it the case that ?

* Is ? Yes, because
* Is ? Yes, because:

1. odd, :

Now:

* because .
* because:

**Proving Recurrence 3**

Prove that is correct.

Above is recursive version of binary search. Iterative version:

Typically, for iterative algorithms, towards correctness, we articulate a *loop invariant*:

Let and be the values of and respectively on input. Just before we successfully enter an iteration of the loop of Line (1), it is true that:

Going back to the recursive version, what is a correctness property?

Given an array that is sorted, non-decreasing, are each on input, returns:

Proof by case analysis:

Case 1: on input: then condition of Line (1) evaluates to **false**, and we correctly return **false** in Line (6). Then, this is either from Line without making any recursive calls, or as the return value from a recursive call from one of Lines or .

For , we first observe that because the only recursive calls are within the block of Line . So, all that remains to be proven is that indeed: .

We prove that by induction on . Base case: . We claim we return within the first recursive invocation. That is, we claim: and , , and .

easy to prove:

is , because then we would have returned in Line .

To prove : we simply exploit:

So, the algorithm is correct if it returns , and .

For the step, we know that on input . So, we returned in some recursive call. So, all we have to prove to appeal to induction assumption: and .

**Proving Master Theorem**

Give a closed form solution for the following recurrence. Assume: is non-decreasing, .

Proposed approach: Inductive “rewriting” of the function . But first: adopt concrete functions wherever we have , or . In this case: adopt for , and for . Now onto the rewriting:

To figure out the power of in that last term:

Power of is the same as the power of inside the . In other words: what is the power of , i.e., for which ? Answer: .

Our next step: Simplify/figure out:

Suppose:

Now subtract one from the other:

When , how do we figure out what is? Answer: then, is:

So, going back to our :

And:

When is ? Answer: .

So, going back to our : first, the case that .

But even before that: rewrite . Because:

So, when . So, in this case:

Onto the other two cases: .

Before we continue: a closer look at :

So: when

So, going back to :

So, if :

And if :

**Proving Greediness**

A picture containing diagram

Description automatically generated

Candidate greedy choice: request with earliest finish time.

Proof strategy: “cut and paste.”

For this problem, we prove two claims in order:

Claim 1: *Suppose for some input of requests, is an optimal (maximum-sized) set of requests which are pairwise conflict-free ordered in increasing finish time. Suppose our greedy algorithm outputs , ordered in increasing finish time. Then, it is true that: for every* .

*Proof.* Note: it must be the case that . And therefore, , i.e., greedy is optimal.

Proof by induction on . Base case: . In our greedy algorithm, we first pick exactly a meeting that finishes earliest amongst all requests. Therefore, immaterial of what is, .

Induction assumption: for , it is true that .

Step: to prove that . We observe:

* – because the set is conflict-free requests, ordered in increasing finish, and therefore, start times.
* – induction assumption.
* Therefore, . Therefore – because after we greedily choose and eliminate all requests that are in conflict, still remains. And our greedy choice is exactly to pick a request that remains that finishes earliest, and we happened to pick .

Claim 2: *Given sets as in Claim 1, cannot exist in* .

*Proof.* By Claim 1, . And because the set is all conflict-free, . Therefore, . So, not in conflict with , and so was available to be chosen after was chosen and all conflicts were eliminated.

Contradiction to the assumption that greedy algorithm terminates only when no more requests available to choose from.

**Graph Algorithm 1**

Graphical user interface, text

Description automatically generated

Time efficiency of

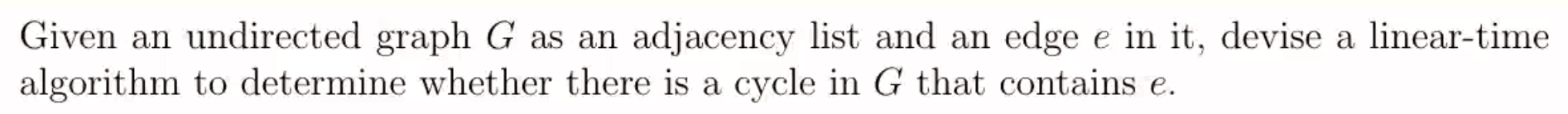
Perhaps a better (more efficiency) approach:

* Visit each vertex as though it is someone’s neighbor.
* Measure its degree.
* Walk its adj list again and inform each neighbor of the degree so they can update their .

Time efficiency:

* We visit each vertex once – Line (4) loop.
* We visit each edge four times – Line (6) and Line (7), we walk each adj list twice.
* So total time: .

**Graph Algorithm 2**



“Go-to” linear time algorithms for graphs: DFS and BFS.

Idea:

* DFS, check if back edge results in DFS tree.
* In fact, edit the explore routine as follows:
  + Keep track of parent in DFS tree.
  + Every time we hit a vertex, check if edge to root of DFS tree, and root is not parent in DFS tree.
  + If yes, immediately output **true**.

**Proving DAG**

Show that the following algorithm to linearize a DAG can be realized in linear time.

*Find a source, output it, and delete it from the graph.*

*Repeat until the graph is empty.*

We assume adjacency list representation of the input DAG.

Suppose we first create a new array, call it of size , where is the number of edges incident in at the start. Can do this in one pass of entire adj list of the graph.

From , we can identify all sources. Suppose we create a list of source vertices, call it . Then, we remove a vertex from and proceed…

**Proving Depth First Search (DFS)**

Prove that DFS on an undirected graph can result in no cross edges.

An edge is a cross edge if and only if: .

Suppose a cross edge, exists after a run of DFS on an undirected graph .

At the time and at all times prior since initialization, .

But that means that in the for loop that immediately precedes , we would have invoked , thereby setting to **true** before the time .

Therefore, we have a contradiction.

**Proving Shortest Path**

Professor F. Lake suggests the following algorithm for finding the shortest path from node to a node in a directed graph with some negative weight edges: add a large constant to each edge weight so all the weights become positive, then run Dijkstra’s algorithm starting at node , and return the shortest path found to node .

Is this a valid method? Either prove that it works correctly or give a counterexample.

Directed graph with weights on edges is: , where , and .

Counterexample, add a constant of to the graph below:

Chart, diagram

Description automatically generated

In the unmodified graph, the shortest path is (), but in the modified graph, the shortest path becomes (). Since the shortest path changes, this is not a valid method.

**Proving Dijkstra’s**

Prove: if we initialize to , and at the end of a run of Dijkstra’s algorithm on with source it is the case that , then there exists a path in .

Contrapositive: if there exists no path in , then at the end of any run of Dijkstra, .

We first observe: the only way can change after initialization is via a call where is incident on , i.e., some .

So proof strategy: induction on number of invocations to that the run of Dijkstra does. Call this number .

If , then this can only be because . Then, there is no path . And as we have not changed from its initial value, at the end of the run of Dijkstra, as desired.

For the step, we consider two cases.

(i) No edge is incident on . Then, we know that no affects , and therefore as desired.

(ii) There exists some . If the last we performed is not on any edge incident on , then is the same as it was after invocations to , and by the induction assumption in that case .

The final (sub-)case: the update was on some , i.e., edge incident on . Then there is no path . Why not? Because if there was, there would be a path to . And therefore, is whatever value it is after invocations to . And by the induction assumption before . Also, again by the induction assumption, before the invocation to . Therefore, after the invocation, which is , .

**Proving Bellman-Ford**

Prove: suppose we run Bellman-Ford on where we do not know whether has a negative weight cycle. Also suppose that at the end of that run of Bellman-Ford, we carry out one more on every . Then: some changes in this additional round of updates for some that is reachable from if and only if there is a negative weight cycle in that is reachable from .

“Only if”: we seek to prove: if changes, this implies that there is a negative weight cycle.

By Claim (2) of Lecture 5(b): if there exists a shortest path from to that is simple, then invocations to on all edges, as Bellman-Ford does, is sufficient for to converge to . Given that invocations to on all edges is not sufficient, this can only be because there is a shortest path that is not simple. And this in turn is true only if there is a negative cycle reachable from .

“If”: we seek to prove: if there is a negative weight cycle reachable from , then there exists some that is reachable from for which the additional round of changes .

An observation: a change to has to be a decrease. Because (repeated) invocation(s) to can only decrease value(s).

Suppose is a negative weight cycle that is reachable from , where .

Proof idea: we know that .

Assume, for the purpose of contradiction, that no changed in the additional round of ’s, for any . Now consider the vertices in the negative weight cycle above.

We first observe for all , it is true that: after the last round of updates.

So now:

But:

To see this:

Because .

So:

This is the contradiction.

This proof is “constructive” – it is saying that must change (decrease) for some vertex in a negative weight cycle reachable from in this additional round of calls to .

**Minimum Spanning Tree 1**

Give an example of a connected undirected such that the set of edges : there exists a cut such that and is an edge of smallest weight that crosses does not form an MST.

Let be that set of edges.

What is the set for our example graph: complete graph with vertices .

.

* Is ? Yes. It is a light edge that crosses the cut: .
* Is ? Yes. Consider the cut .
* Is ? Yes. Consider the cut .

So, we are done. Because is not an MST of that .

Because that is not acyclic, i.e., not a tree.

**Minimum Spanning Tree 2**

Professor Sabatier conjectures the following converse of what we say under “now the general approach” on page 3 of Lecture 6a:

Let be a connected undirected graph. Let: (i) that is included in some MST of , (ii) be any cut of that respects , and (iii) also be included in some MST of . Then, is an edge of smallest weight that crosses the cut .

Show that the professor’s conjecture is not necessarily true.

Let , , and , .

Now let . Then the cut respects .

And is not a light edge that crosses that cut but is in a (the) MST of the graph.

**Min-Cut and Optimal Substructure**

Consider the min-cut problem: given as input an undirected graph , what is the minimum number of edges that cross any cut where ?

Bob claims that the problem has the following optimal substructure. Given a cut that is a min-cut for , then is a min-cut for , where we get from by removing and all edges incident on it, provided both and are non-empty.

Refute Bob’s claim.

Examples:

Chart, bubble chart

Description automatically generatedChart, bubble chart

Description automatically generated

Original min-cut , and indeed, for this min-cut, it turns out:

is indeed a min-cut for for this .

Diagram

Description automatically generated with medium confidence

Counterexample: In this example, the min-cut is . However, if the candidate vertex is removed, the graph can be shifted such that the min-cut now becomes . Hence, this is a valid counterexample, and Bob’s claim is refuted.

**Optimal Substructure**

The interval-scheduling problem from “Proving Greediness” possesses optimal substructure. What is it, and how do we exploit it to realize an algorithm?

We are given as input a set of requests , where each such that and .

For single-source shortest-paths: if is a shortest-path from to , then the sub-path is a shortest path from to .

We could ask: suppose is an optimal sequence of requests that are non-conflicting such that . Is there anything I can say about the optimality of ? More specifically, is it an optimal solution to a sub-problem?

I think: the answer is yes. A sub-problem for which has to be an optimal solution: suppose in the input, the requests are ordered by non-decreasing finish time. Then: has to be an optimal solution to all requests that end at or before .

In fact: we can “lop off” or “eat into” optimal solution from both directions.

Specifically:

* Assume input set of requests are sorted non-decreasing by finish time. That is: .
* Now: suppose is the max # requests I can schedule that start at or after and end at or before .
* Also denote as the set of requests that start at or after and end at or before .

Then:

Our final solution: , where we introduce fictitious requests with , .

**k-ary Search**

In binary search, we split the input sorted array into two pieces, each of size , and recursively search on one of those pieces.

Alice proposes k-ary search, in which we split the array into pieces, each of size . What is the worst-case time-efficiency of k-ary search as a function of ?

Inspired by the eggs-building problem, Alice wonders whether setting yields a more efficient algorithm than binary search. Does it?

Chart, timeline, bar chart

Description automatically generated

Recurrence for binary search: .

For k-ary search, recurrence for worst-case time-efficiency: . To solve the recurrence:

Where we figure the last term as follows: we ask for what is ? Answer:

So, if we set , then:

And if we do binary search, . And , and . So, setting yields a strictly worse performing algorithm than setting .

**Binary Search**

Carry out an expected- (or average-) case analysis of the time-efficiency of binary search.

First off, we should distinguish a successful (binary) search from an unsuccessful search. Because expected-case time-efficiency of an unsuccessful search is just .

For a successful search, the time it takes depends on the item we are looking for. Assume: (i) every item in array is distinct, and (ii) every item in the array is equally likely to be searched for.

Now, if is a random variable that is the number of comparisons or # iterations or # recursive calls we perform before we find the item we seek is below. As simplification, assume that we have items in the array, for some positive integer .

Timeline

Description automatically generated