Amortized Time Complexities & Brute Force Algorithms

Suhas Arehalli

COMP221 - Spring 2024

1 Tangent: Amortized Time Complexities

- In Skiena, you saw that when analyzing the behavior of array-based stacks, queues, and lists, we must occasionally double the size of the underlying array
  - Stacks, queues, and lists have *unbounded* length, while arrays have a *fixed, finite* length.
  - To maintain the illusion that the underlying array is unbounded, whenever we would go past the bounds on our array, we secretly create a new array with double the length and copy the contents of our array over.
  - This is an $\Theta(n)$ operation, as opposed to the $\Theta(1)$ inserts we get when we don't have to expand the array.
  - This means that the worst-case time complexity of an insert into an ArrayStack or ArrayQueue is $\Theta(n)$!
- However, we know that this doubling is fairly rare — most of the time we can do inserts in $O(1)$ time!
  - It feels misleading to have our worst case be $O(1)$ when we will only rarely have to double the array's size
  - Also note that this behavior has to do with the size of the underlying array, rather than getting a bad input. Average-case analysis vs worst-case analysis doesn't help us, because both only consider the behavior of a single call to insert, differing only in how bad the inputs to insert are.
- To counteract the misleading nature of the worst-case analysis, we can consider a new kind of time complexity analysis: *Amortized* time complexity
  - Here, we analyze the average behavior of an operation like insert *over time.*
  - More formally, we can think about this as analyzing a sequence of inserts that run one after another, and averaging their asymptotic behavior.
- **Example:** Inserts into an ArrayStack/ArrayQueue

  Suppose our Array is initialized with size m. This means that we double, and incur an $\Theta(n)$ cost for the insert of the $m + 1$th item, $2m + 1$th item, the $2^2 + 1$th item, and so on. Since the cost has order proportional to the size of the array at the point of doubling, each of these will incur a cost proportional to m, $2m$, 2^2m, and so on. If we do n inserts, this means that our growth function is bounded by

$$\sum_{i=1}^{n} k + 2 \cdot \sum_{i=1}^{\log n} 2^i m$$

Note that the 2 outside the sum represents the cost of each copy operation (one memory access to retrieve the previous value, and one to assign it to it's new spot. In practice, it doesn't change the analysis, but still good to note!). The first sum is the normal constant cost of each of the inserts (some constant number of operations to update pointers and place the inserted element into the right array).

Simplifying:

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} k + 2 \cdot \sum_{i=1}^{\log n} 2^i m \\ &= kn + 2m(2^{\log n + 1} - 1) \\ &= kn + 2m(2n) - 2m \\ &= kn + 4mn - 2m \end{aligned}$$

Remember that this is the analysis of n consecutive inserts! Thus, on average, our time complexity for a single insert is $\frac{1}{n}$th of this!

$$\begin{aligned} \frac{1}{n} f(n) &= k + 4m - \frac{2m}{n} \\ &\in \Theta(1) \end{aligned}$$

Since m is a constant (our initial array size).

- Remember that what amortized $\Theta(1)$ worst-case time complexity means is that after n operations our worst-case time complexity is $\Theta(n)$ (i.e., it's the same as it would be if each call was worst-case $\Theta(1)$!). We can make no guarantees as to whether any particular insert will be $\Theta(1)$ or $\Theta(n)$, and so non-amortized time complexities are still stronger guarantees than amortized ones.

2 Brute Force Search

- Our first (and simplest) algorithm design strategy.
- Idea: Iterate through every possible solution and check if it's correct.
- To solve any problem with Brute Force Search, we just need two things:
  1. A way to iterate through the *solution space* of our problem.
  2. A way to verify whether a candidate solution is correct.
- When we have those two, we can follow the brute force search template, outlined in Alg. 1.

Algorithm 1 A template for brute force solutions

```
for $candidate \leftarrow SolutionSpace$ do
    if $candidate$ is the solution then
        return $candidate$
    end if
end for
return solution doesn't exist
```

- Example: Linear Search.

  Consider LinearSearch (Alg. 2).

  Correctness for a search in an array is defined by cases: If we return an index i, the algorithm is correct is $A[i] == e$. If we return $NULL$, then we're correct if $e \notin A$.

  To construct a brute force solution to this search problem, we first observe that our solution space consists of indices $\{1, \dots, N\}$ and $NULL$. Lets set aside $NULL$ for a moment. We are then given a direct way to verify the correctness of a candidate solution: If we return some index i, our problem specification tells us that this is correct iff $A[i] == e$. Simply substituting that solution space and verification procedure into the brute force template gets us most of linear search!

  What remains is the fact that instead of saying that a solution doesn't exist, we return $NULL$. There are two ways of looking at this, both of which get you to the right intuitions: First, we can reframe our problem specification to the following:

  > *A search algorithm over an array A returns an integer i such that $A[i] == e$, or $NULL$ if no such i exists.*

  This is equivalent to the previous definition (having no i such that $A[i] == e$ is the same as saying $e \notin A$!), but now frames the $NULL$ return value as a *solution doesn't exist* value, and now LinearSearch perfectly matches the brute force template!

  An alternative way of seeing this is to treat that last return as (effectively) a final iteration of our loop. Once we've gone through every possible index, we have to check the last element in our solution space, $NULL$. Verifying whether $NULL$ is correct is, by our definition of correctness, checking whether $e \notin A$. However, as we just saw, this is equivalent to checking whether $A[i] \neq e, \forall 1 \leq i \leq N$. But, rather than doing that, we can apply a small optimization: We only reach the $NULL$ condition if we've already verified that exact condition, so we can avoid that complex if-statement and just return $NULL$.

  Regardless of which approach makes more sense to you, we can hopefully see that LinearSearch is almost a prototypical brute-force solution to the search-over-arrays problem!

2.1 Correctness of Brute Force algorithms

- Proofs of correctness for brute force algorithms are, as the template might suggest, relatively easy.

Algorithm 2 A template for brute force solutions

```
function LinearSearch(Array $A$, target $e$)
    for $i \in 1$ to $N$ do
        if $A[i] == e$ then
            return $i$
        end if
    end for
    return $NULL$
end function
```

- If we've constructed our verification method correctly, we should be able to show that for any return value from inside the for-loop, the value returned is correct (otherwise, we shouldn't have gone inside the if-statement!).
- If we reach the *no solution exists* return statement, then if we've constructed our solution space correctly, we should be able to show that every potential solution has been shown to be incorrect!
- The challenging part is usually ensuring that you've chosen the right solution space (you aren't skipping any possible solutions!) and that your verification method works (you don't mistakenly classify wrong candidate solutions as correct or vice-versa!).
    - For something like search over an array, this is straightforward. But for more complex problems, this can be tricky!
- Because brute-force algorithms are comparatively easy to prove correct (and thus to write correctly!), when the time complexity is not prohibitive, these kinds of algorithms are sometimes preferred for situations where ensuring correctness is critical.

2.2 Runtime Analysis for Brute Force Algorithms

- As you might imagine, Brute-Force algorithms are often very slow.
    - Because this design strategy often leads to minimally-clever, slow algorithms, people often colloquially call any slow, unclever algorithms as "brute-force." For the purposes of this class, I'll try to exclusively use the term to refer to this particular design strategy.
    - Often referring to this design strategy as *Brute Force Search* can disambiguate
- Applying the principles of Big-Oh analysis to our template for brute force solutions, we can learn a few things:
    - Suppose we have a $\Theta(f(n))$ algorithm for verifying a candidate solution's correctness and we know the solution space's size is $\Theta(g(n))$.
    - Worst-case, we go through every element in the solution space, each time checking the candidate items correctness. Thus the $\Theta(f(n))$ verification algorithm runs $\Theta(g(n))$ times.
    - With a bit of big-Oh math (a good exercise if you want some extra practice!), you can show that this means our algorithm will be $\Theta(f(n)g(n))$!

  - In practice, verification will often be fairly fast (polynomial time, say), but our solution spaces can very quickly get unwieldy! This is what makes brute-force solutions something we want to avoid!
  - However, in some cases, there is nothing more clever we can do. For instance, in the general case for search over an array, we can do no better than LinearSearch! There is no additional information about where the target element might be that lets us avoid checking every index!
    - A fairly general argument might go something like this:

      > Suppose we find an algorithm faster than $\Omega(n)$ for our search-over-arrays problem. This means that we cannot have checked whether $A[i] == e$ for all $1 \leq i \leq N$, since checking each i would result in n operations. This means that for any input array A, $\exists j$ such that we don't check whether $A[j] == e$. No assumptions prevent $A[j] = e$ and $A[k] \neq e, \forall k \leq j$, and so we can't guarantee that our algorithm is correct!

    - The key, sneaky line in this argument is that no assumption prevents the indices we don't check to be the ones that we're looking for. As long as this is true, we have to check every index!
    - However, if we can make some stronger assumptions, we can do better. For instance, if we know that the input array is, for instance, *sorted*, this argument doesn't hold! We can do better (i.e., BinarySearch!).

2.3 Brute-Force Algorithms in the Wild

2.3.1 Password Cracking

- Sometimes, we can take advantage of the fact that some problems don't have (known) algorithmic approaches faster than brute-force. This is true of cryptography, where people use a lot of fancy math (Number Theory, mostly) to concoct problems that can only be solved by brute force.

- Consider this problem specification:

  > *Given a function f that maps a password string s to either true or false, find some String s of length k with $m \leq k \leq n$ such that $f(s)$ is true*

- This is the exact problem you face as a hacker (or as a red-team cybersecurity professional testing the security of a system) with access to unlimited attempts at guessing a user's password.
  - In general, websites will try to lock you out after too many password attempts. This is good practice!
  - This situation does arise in practice though, but only after security has been breached in a different way! Even if a website's servers have been hacked, password databases aren't (or, shouldn't be!) stored as *plaintext*. Instead, they're stored as hashes!
  - That is, the passwords are put in cryptographic hash functions and the output is stored, under the assumption that these hash functions are difficult to *invert*. Knowing the password lets you compute the hash, but knowing the hash doesn't let you know the password!

  - The ideas of hashing for hashmaps and hashing for cryptography are deeply related, but slightly different (we want cryptographic hash functions to have properties we don't really care about for constructing a hash table).
  - What's important here is that if you are a hacker with access to password hashes, you're left with this exact problem specification: You have a hash function that will tell you when you've found the right password, and you can run it as many times as you want!

- Our solution space here is all strings of length k between m and n. How big is this space? If there are l different characters, then

$$= \sum_{i=m}^{n} l^i \tag{1}$$

  but this expression really doesn't capture how large this can be. First, let's *underestimate.* If we ignore every term in the sum but the last, we can see that $\sum_{i=m}^{n} l^i \geq l^n$ which means that this algorithm is $\Omega(l^n)$ (Exponential time! Finding that the upper bound is exponential as well to show it's $\Theta(l^n)$ is also left as a quick exercise).

  But to be super clear, let's underestimate again. Suppose passwords can only be composed of letters of the English alphabet. There are 26 letters, and so our lower bound is 26^n. If $n = 10$, checking every password of length n would take $26^{10} \approx 1.4 \cdot 10^{14}$.

  If checking each password takes 1 millisecond, checking every password takes $1.4 \cdot 10^{11}$ seconds. That is about 4,436 years. And this is a vast underestimate in a lot of ways!

  ASCII has 128 characters. If we consider passwords of length between 8 and 12, we have

$$\begin{aligned} \sum_{i=8}^{12} 128^i &= 19495118728336243746144256 \\ &\approx 1.9 \cdot 10^{25} \end{aligned}$$

  What we may have overestimated is the time it takes to run a hash function. Let's be super generous, and say that we can verify one candidate password in a nanosecond. That is, we can check 1 *billion* hashes per second (maybe with a fast hash and a bit of parallelization)!

  This means it takes approximately $1.9 \cdot 10^{16}$ seconds to check every one. This is over 618 million years. This is about $\frac{1}{7}$th of the age of the Earth.

  This is why our passwords stay safe!

- I've framed this in terms of cracking password hashes, but this also applies to a bunch of other cryptographic problems (most notably, public-key cryptography!).

- The general idea is that, for various reasons, we can't make it impossible for malicious actors to be guarantee access to secure information we transfer over the internet. However, if we're clever, we can manipulate the time complexity of the way in which those actors can gain access to that information. Above, we saw that since we can't prevent malicious actors from being able to verify the correctness of a password, we can't stop them from using a *brute force attack* to find the correct password. However, we can increase the size of the solution space (i.e., the number of legal passwords) to make it take longer than a human lifespan by several orders of magnitude for that malicious actor to crack a password.

- A fair amount of cryptography/security is making sure we can't do better than a brute force attack. Sometimes, a sloppy implementation, new mathematical results(!), or some exploitation of human behavior can allow you to avoid checking every possible password!

- One thing to pay particular attention to is the fact that in order to make the solution space grow *exponentially*, we only make users suffer a *linear* cost: Increasing the length of legal passwords by 1 to n adds k^n new passwords to check! This is often referred to as a *combinatorial explosion*!

2.3.2 Chess & Tablebases

- One common variant of brute force search problems involves games with *perfect information* and no randomness.

- A game with perfect information is one like chess: There is no hidden information, and all players know all of the options available to each other. Other games include tic-tac-toe, connect four, set, etc.

- Because there is not randomness and every player could, theoretically, calculate every possible sequence of moves that leads to the end of the game, we're often interested in whether the game is *solved*. That is, if we can determine the optimal strategy for each player (typically, one where one player is guaranteed to win, or either player can guarantee a draw).

- Much like the cryptography example, we often want these games to *not* be practically solvable. Otherwise, playing a game is simply a test of whether you or your opponent know the optimal strategy (i.e., every game of tic-tac-toe should be drawn!).

- Chess survives as a competitive game because the game "suffers" from combinatorial explosion when one tries to compute possible game states.

  - Each player typically has a large number of potential moves available to them: In most board states, nearly every piece on the board has one or more moves, meaning that we have to consider a future moves that stem from each of those hypothetical moves.
  - We can analyze this situation in terms of a *decision tree*. Each node represents a state of the game board and the children of each node correspond to the board after the active player makes a legal move. From this, we can see that each node has k children, where k is the number legal moves the active player can make. If the game state is won or drawn, $k = 0$, and we know which player won.
  - As long as the average k is greater than 1 at each level of the tree, the number of nodes grows exponentially with each additional level!

- Because of this combinatorial explosion, we have to use trick to analyze chess states: heuristics about piece value, approximate methods (branch & bound, etc.), etc.

- **OR**, we try and brute force what we can!

  - Chess players are often concerned with whether a particular endgame (a stage at the end of the game where most of the pieces are off the board) are winnable.

  - Since there aren't as many pieces on the board (i.e., ks, our *branching factors*, are relatively small) and games are close to over (the number of levels in our decision tree we need to analyze before we get to leaves is small), we can brute-force analysis!
  - Note that one must still consider the number of starting positions: How many legal placements of 6, 7, or 8 pieces are there on a standard chessboard?

- The results of these brute-force analyses are called *(Endgame) Tablebases*.
- As of right now, tablebases typically can only cover positions where 6 or 7 pieces remain on the board, with work being done to complete a tablebase for 8 piece positions.
- For reference, 7-piece tablebases must consider 423,836,835,667,331 positions, and take up roughly 140 terabytes of memory![1]

[1] This is in reference to the Lomonosov Tablebase, which analyze the game state until one player wins. Some tablebases are smaller, but use heuristics and shortcuts to avoid some computation (i.e., marking game states where a pawn is promoted as won for the player that promotes). The Lomonosov table claims to analyze all sequences until a player checkmates or a draw is guaranteed, which is more in the brute-force spirit.