Algorithms

Theresa Migler-VonDollen
CMPS 5P
Write a Python function that accepts a list of numbers and a number, x. If x is in the list, the function returns the position in the list where that number appears. If x is not in the list, the function returns the value -1.

- Note that this is very similar to the built-in function `index`, except that it handles exceptions.
- However, we are interested in how the search is actually performed.
Suppose that you are given a list of 1,000 numbers (unsorted) and you are asked to find a particular number.

- How would you do this?
  - Scan the whole list.

This strategy, scanning a list one by one, is called linear search.

This strategy works well for modest-sized lists.

Both in and index use linear searching algorithms.
Now suppose that the list of number is sorted (low to high).

Do we still need to scan the list one by one?

- How would you do it?
- You could scan until you find a number that is greater than $x$.
- On average this will save half the time from our previous algorithm.
How would you find $x$ in a sorted list of 1,000 numbers?

- You would probably look at the middle number in the list and compare it with $x$, then throw half the list away.
- This strategy is called *binary search*. At each step, we divide the list into two parts and throw away one of the parts.
Since \( x \) could be anywhere in the list, we start with \( low \) as the first location in the list and \( high \) as the last location in the list.

This algorithm will look at the middle element in the range and compare it to \( x \).

- If \( x \) is larger than the middle element, we move \( low \) to be the location of the middle element.
- If \( x \) is smaller than the middle element, we move \( high \) to be the location of the middle element.
Write a Python function, using binary search, that accepts a list of numbers and a number, $x$. If $x$ is in the list, the function returns the position in the list where that number appears. If $x$ is not in the list, the function returns the value -1.
Comparing Algorithms

- Which algorithm is better?
  - Which algorithm is easier to understand and implement?
  - Which algorithm runs faster?
- Intuitively, we might expect linear search to work well on small lists while binary search would work better on longer lists.
  - We could test empirically.
  - We could argue abstractly.
To argue abstractly, we need to test the number of "steps".

For searching, the difficulty is determined by the size of the collection - it takes more steps to find a number in a collection of a million numbers than it does in a collection of 10 numbers.

How many steps are needed to find a value in a list of size $n$?

In particular, what happens as $n$ gets very large?
Comparing Algorithms

- Linear search:
  - If the list has size $n$, we would have to loop through at most $n$ items.
  - The amount of time required is linearly related to the size of the list.
  - This is what computer scientists call a \textit{linear time algorithm}.

- Binary search:
  - If the list has size $n$, we would need to loop through at most $\log_2(n)$ items.
  - This is what computer scientists call a \textit{log time algorithm}.
The logarithmic property can be very helpful.

Suppose you have a New York City phone book with 12 million names.

- You could walk up to a random New Yorker (that is listed in the phone book) and try guessing her name.
- All you are allowed to do is ask if her name is alphabetically before or after one that you find in the phone book.

How many guesses will you need?

- \( \log_2 12000000 \approx 24 \)
- If you had used linear search it would be closer to 6 million!
We mentioned earlier that Python uses linear search in its built-in searching methods. Why?

- Binary search required the data to be sorted.
- If the data is unsorted, it must first be sorted, then searched.
The basic idea between the binary search algorithm was to successfully divide the problem in half.

This technique is known as a *divide and conquer* approach.

Divide and conquer divides the original problem into subproblems that are smaller versions of the original problem.

In the binary search, the initial range is the entire list. We look at the middle element... if it is the target, we’re done. Otherwise, we continue by performing a binary search on either the top half or bottom half of the list.
Write a Python function that calls itself to perform binary search.

- This version has no loop!
A description of something that refers to itself is called a recursive definition.

Unlike English, where you cannot use a word to define itself, in mathematics recursive definitions are common.

An example of a recursive mathematical definition, revisit the factorial:

- We know that $n! = n(n - 1)(n - 2) \ldots (3)(2)(1)$.
- But we could also say: $n! = (n)((n - 1)!)$
Is this definition of factorial circular?
No, because there is a base case: 1! = 1

Write a Python function that recursively computes the factorial of a given number.
Recursive Problem-Solving

- Good recursive definitions have these two key characteristics:
  - There is one (or more) base case for which no recursion is applied.
  - All chains of recursion eventually end up at one of the base cases.
- The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.
Python lists have built in methods that can be used to reverse a list.

Suppose we wanted to reverse a string recursively?

Write a recursive Python function that takes in a string as a parameter and returns the string with all characters in reversed order.

Don’t forget the base case! Here it is the empty string.
An anagram is the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once.

An example is “silent” and “listen”.

Anagram formation is a special case of generating all permutations (rearrangements) of a sequence, a problem that is seen frequently in mathematics and computer science.
Write a Python function that, given a word, returns a list with all possible permutations of that word.

- Hint: Suppose we already had all permutations of the word without the first letter. How could we use this to find all permutations of the whole word?
- How many permutations of a word of length $n$ are there?
Recursion versus Iteration

- There are similarities between iteration (looping) and recursion.
- In fact, anything that can be done with a loop can be done with a simple recursive function.
- Some problems that are simple to solve with recursion are quite difficult to solve with loops.
- Often, recursive solutions are more efficient than their iterative counterparts. But not always...
Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13, 21, ...

Write an iterative Python program that takes an integer, \( x \), as an input and returns a list with the first \( x \) Fibonacci numbers.

Write a recursive Python program that takes an integer, \( x \), as an input and returns a list with the first \( x \) Fibonacci numbers.
Recursion versus Iteration

- Which program runs faster?
- The recursive algorithm is extremely inefficient because it performs many duplicate calculations.
- Recursion is another tool in your problem-solving toolbox.
- Sometimes recursion provides a good solution because it is more elegant or efficient than a looping version.
- At other times, when both algorithms are quite similar, the edge goes to the looping solution on the basis of speed.
Suppose you are given a list of numbers in arbitrary order and your task is to rearrange them in increasing order.

How would you do this?

One simple (and correct) method is to look through the entire list and find the smallest element, then look through the entire remaining portion of the list for the second smallest element....

This algorithm is called selection sort.

Is this an efficient algorithm?
Write a Python program that takes as an input a list of numbers and returns the same list in sorted (increasing) order using the selection sort algorithm.
Mergesort

- Suppose that you and your friend needed to sort a deck of cards quickly.
- You might split the deck in half, and each sort your half, then combine the two decks.
- This is a divide and conquer approach.
- This algorithm is called mergesort.
- Practice with an example on the board.
Write a Python program that correctly merges two sorted lists.

Write a Python program that takes as an input a list of numbers and returns the same list in sorted (increasing) order using the mergesort algorithm.
Selection Sort versus Mergesort

- Which algorithm should we use?
- Certainly, the time it takes to sort the list depends on the size of the list.
- The question is: How many steps does each algorithm require as a function of the size of the list being sorted?
- Suppose that we have $n$ items in our list.
For the first step, in the worst case, we need to look through all $n$ items.

For the second step, we might need to look through all remaining $n - 1$ items.

The total number of iterations is:

$$n + (n - 1) + (n - 2) + (n - 3) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

So the running time is proportional to $n^2$.

We call this a **quadratic** algorithm.
Running Time Analysis of Mergesort

- For Mergesort, the place where the real sorting occurs is in the *merge* function.
- Practice sorting [3, 1, 4, 1, 5, 9, 2, 6].
- Starting from the top, we have to copy $n$ values into the second level.
- From the second to the third levels, the $n$ values need to be copied again.
- Each level requires copying $n$ values....How many levels are there?
  - $\log_2 n$.
- Therefore the total work required is $n \log n$. This is called an $n \log n$ algorithm.
Comparing Quadratic and $n \log n$ Algorithms

![Graph comparing 'selSort' and 'mergeSort'](image)
There are some problems for which there is no known fast algorithm.

The best we can seem to do on some problems is an exponential algorithm.

What do computer scientists do when they encounter such problems?