Algorithms Handbook

Shahzeb Asif

August 21, 2014

References

The Algorithm Design Manual

Most of the following notes are based on The Algorithm Design Manual by S.S.Skiena.

Others

- Backtracking
- LCS
- LIS

Introduction

What is an Algorithm?

An **algorithm** is a sequence of steps designed to solve a problem.

Each **problem** has its own **instances** which are specific and apply to a specific situation.

Communicating Algorithms

Algorithms can be expressed in many different ways.

Pseudocode balances the detail and verbosity of code and the ease of understanding of plain English to represent an algorithm.

All algorithms need a clearly defined problem. A badly defined problem will likely give a badly designed algorithm.

Correctness

A large part of designing and using algorithms is verifying that they are actually correct.

Formal **proofs** can get very complicated and are difficult to understand without a math background.

But there are other ways of showing that an algorithm is correct besides a rigorous mathematical proof. Note that even obvious ”algorithms” can be very incorrect.

Heuristics are similar to algorithms, in that they attempt to solve problems, but they do not guarantee an optimal solution. Heuristics appear in situations where no algorithm exists.

Incorrectness

It is often easier to demonstrate that an heuristic is incorrect than correct.

Counter examples are a specific input to an algorithm that gives an incorrect output.

Counter examples should be verifiable and simple. It should be easy to explain why the counter example fails and what the correct output should have been.

Useful skills in finding counter examples:

- Small test inputs.
- Exaustive test inputs.
- Test assumptions.
- Test for ties.
- Test for corner cases.

Beware that failing to prove incorrectness does not imply correctness.

Induction

Induction is a fairly simple tool to help prove the correctness of iterative and recursive algorithms.

In induction, we assume correctness for case k and then prove correctness for case $k + 1$.

Summation

$\sum$ is often used in algorithm analysis.

Two types of summations cover a large number of algorithm analysis.

- **Arithmetic progressions** have changing bases with a constant exponent.

$$\sum_{i}^{n} i^p = \Theta(n^{p+1}) \tag{1}$$

- **Geometric series** have changing exponents with a constant base.

$$\sum_{i}^{n} a^i \tag{2}$$

Modeling

Modeling the problem is necessary to see it in its general form.

Modeling a specific instance of a problem may reveal the problem to be a familiar graph problem or a tree.

There are a few useful structures that can help model a problem:

- Trees
- Graphs

- Points
- Polygons
- Strings

Analysis

The most common form of analysis is to talk about an algorithm's complexity.

Complexity

We use the Big O notation to compare algorthm performance.

We only care about three cases:

- Worst case. Represented by $O(g(n))$.
- Average case. Represented by $\Theta(g(n))$
- Best case. Represented by $\Omega(g(n))$

There are a few different classes of complexity that cover most algorithms.

- Constant - 1
- Logarithmic - $log(n)$
- Linear - n
- Superlinear - $n * log(n)$
- Quadratic - n^2
- Cubic - n^3
- Exponential - c^n
- Factorial - $n!$

The relation between the complexity above can be thought of as follows:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n * log(n) \gg n \gg log(n) \gg 1 \tag{3}$$

Finding the complexity of combinations of algorithms or just the combination of functions requires adding or multiplying functions.

Addition

The complexity of the result of two added functions is the complexity of the most expensive function.

This is to say that the Big Oh of $2^n + 3n + 4$ is $O(2^n)$ or exponential even though it contains a linear and a constant element.

Multiplying

Multiplying functions with constants results in the same complexity, i.e. $c*n = O(n)$ for all positive values of c.

The result of multiplying functions by other functions makes the result have a complexity analogous to the product of the functions. For example:

$$f(n) * g(s) = h(t) \tag{4}$$

$$O(h(t)) = O(f(n) * g(s)) \tag{5}$$

Logarithms

Logarithms are an important tool for analysing complexity. It's difficult to get an intuition for logs but remember that they are just constructs to help solve equations with exponents.

Here are the log rules:

$$b^x = y \iff log_b y = x \tag{6}$$

$$lnx \iff log_e x \tag{7}$$

$$log_a(x) + log_a(y) = log_a(xy) \tag{8}$$

$$log_a(x) - log_a(y) = log_a(\frac{x}{y}) \tag{9}$$

$$log_a(b) = \frac{log_c(b)}{log_c(a)} \tag{10}$$

Data Structures

Data structures are what allow a program to store information in some format. Different data structures are made for different purposes.

Using an appropriate data structure could dramatically speed up a program.

There are two main types of data structures: contiguous and linked.

- **Contiguous** data structures are stores as one slab of memory. An example of these are arrays.
- **Linked** data structures are isolated pieces of memory joined by pointers. An example of these are linked lists.

Arrays

Arrays are the poster boy for contiguous data structures.

We can easily allow indexing into an array because we can index in constant time using pointer arithmetic.

Arrays also allow **space efficiency** because all the data is stored as just one large block and no extra memory is wasted.

Arrays have **memory locality** because all the data is close to each other. This allows the processor to speed up execution by using the cache.

Arrays are not good for dynamic length because there is no easy way to consistently make an array longer in execution.

But **dynamic arrays** are actually possible by creating a new array, double the size of the previous one, and copying all the previous data to the new array if you run out of space. The disadvantage with this approach is that sometimes indexing into an array may take longer than constant time.

Lists

Because of their nature, to talk about lists in detail, we must talk about implementing them in `C`.

Lists are created by pointers. A list in `C` may be created with the following struct:

```
typedef struct list {
    uint8 item;
    list *next;
}
```

A List in `C`

This `struct` allows you to store a `uint8 item` and a pointer to the next item. This was you can easily transverse an entire list by following the `next` pointer.

Now we just need to define where a list starts and ends. If we store a pointer called `head` that points to the start of the list, we can easily access the list. As for the end, we can have the `next` pointer point to `NULL` for the last item. We could also store a pointer to the last item for convenience.

Because lists are independent pieces of memory linked with pointers, we can easily make a list longer or shorter.

To make a list longer, we would simple allocate a new list item using the `struct` above and then attach it to the last item in the list.

To make a list shorter, we can remove any item from the end very easily and then free the memory. We can also remove any item in between but that's slightly more tricky.

For these advantages, lists do have their downsides:

- Lists use more memory to store data than an array.
- Lists take longer to transverse than an array.
- Lists may not be cache optimized on allocation.

The `struct` above implements a **singly-linked** list. We can have **doubly-linked** lists as well. A doubly-linked list has pointer to the next and the previous item.

Stacks

Stacks can be thought of as a stack of plates. We can put more and more plates on top but we can't remove the 5th plate from the top without removing the top four first.

For this reason, we consider stacks to be a LIFO data structure. **LIFO** stands for last-in first-out. The last item to be put on the stack must be the first item to be pulled off.

Queues

Queues are very similar to stacks but with one key difference. Queues are FIFO. **FIFO** stands for first-in first-out.

An example of these might be a job list. The first job on the list has to be completed first.

Dictionaries

A real dictionary holds a definition for every word. The dictionary data structure holds a key and a corresponding value for each key.

We can perform many operations on a dictionary:

- Search
- Insert
- Delete
- Max/Min
- Predecessor/Successor

The complexity of these operations depends on the implementation of the dictionary.

Criteria

Before we discuss implementations, we must create a criteria to compare them all on.

The five items we'll use for dictionaries are:

- **Search**
  - The time it takes to search for a specific item.
- **Insertion**
  - The time it takes to insert a new item.
- **Deletion**
  - The time it takes to delete an item but the item's location is known in a list or unsorted structure.

- **Max/Min**
  - The time it takes to find the max or min element.
- **Pred/Succ**
  - The time it takes to return the next or last item in a sequence, e.g. `succ 8 = 9`. Note that we know the location/value of the initial item.

Array

We can use an array to create a dictionary. But we have two options: sorted or unsorted array.

Operation	Unsorted	Sorted
Search	$O(n)$	$O(log(n))$
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$ or $O(n)$*	$O(n)$
Max/Min	$O(n)$	$O(1)$
Pred/Succ	$O(n)$	$o(1)$

Table 1: Dictionary Array Implementations

In an unsorted array:

- Search takes linear time because we need to compare each item.
- Insertion takes constant time because we can just append new data. This assumes that we have space at the end of the array.
- *Deletion can take constant or linear time.
  - It takes constant time if we delete an item and then replace its place with the last item and store the length of the array.
  - It takes linear time if we decide to move all elements up to fill a hole left by deletion.
- Max/Min both take linear time because we need to do a full-scan of the array.
- Pred/Succ in an unsorted array take linear time. The easiest way to see this is to thing of Pred as finding the min after removing a min. This means that we find the min twice and remove an item once. That is still a linear operation.

In a sorted array:

- Search takes logarithmic time due to binary search.
- Insertion takes linear time because we have to find the right spot to insert the item and then shift all the items to make space. Finding the right spot is logarithmic but shifting the items is linear so the whole operation is linear.
- Deletion takes linear time with similar reasoning to insertion.
- Max/Min take constant time because we just use the first or last item.
- Pred/Succ takes constant time because we just return the next or previous item in the array.

Lists

We can also use lists to create a dictionary but this has its own trade-offs.

Note that for deletion, the item to be deleted is passed in as an argument.

Operation	Unsorted	Sorted
Search	$O(n)$	$O(n)$
Insertion	$O(1)$	$O(n)$
Deletion	$O(n)$	$O(n)$
Max/Min	$O(n)$	$O(1)$
Pred/Succ	$O(n)$	$O(1)$

Table 2: Dictionary Singly-Linked List Implementations

For a singly-linked unsorted list:

- Search takes linear time because we compare every item.
- Insertion takes constant time because we can just append an item.
- Deletion takes linear time because we need to fix the `pred` of the deleted item which takes linear time to find.
- Max/Min take linear time because we need to search the entire list.
- Pred/Succ take linear time for the same reasons as an unsorted list.

For a singly-linked sorted list:

- Search takes linear time because we still need to run through each item.
- Insertion takes linear time to find the right spot.
- Deletion takes linear time to fix the `pred` item.
- Max/Min take constant time because we just return the `head` and `tail` of the list.
- Pred take linear time because it is a singly-linked list. Succ only takes constant time.

Operation	Unsorted	Sorted
Search	$O(n)$	$O(n)$
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(1)$
Max/Min	$O(n)$	$O(1)$
Pred/Succ	$O(n)$	$O(1)$

Table 3: Dictionary Doubly-Linked List Implementations

For a doubly-linked unsorted list:

- Search takes linear time because we still transverse the list.

- Insertion takes constant time because we just append the item.
- Deletion takes constant time because fixing the previous item is constant now.
- Max/Min take linear time to transverse the list.
- Pred/Succ take linear time similar to an unsorted array.

For a doubly-linked sorted list:

- Search takes linear time to actually transverse the array.
- Insertion takes linear time to find the right place to insert.
- Deletion takes constant time because a pointer to the item to be deleted is passed in and constant time to fix the pointers.
- Max/Min take constant time because we just need the first and last item.
- Pred/Succ take constant time because this is a doubly-linked list allowing easy access to the previous items.

Binary Search Trees

Binary search trees allow us to do fast search and fast update.

A binary search tree is just a structure with two branches coming out of it. We can call the current node the parent. The parent's left branch has items lower than parent and the parent's right branch has items higher than parent.

We can implement a BST (binary search tree) in `C` using the following struct:

```
typedef struct tree {
    uint8 value;
    struct tree *left_node;
    struct tree *right_node;
    struct tree *parent_node;
}
```

A Tree in `C`

Note that we also store a pointer to the parent.

Because of the nature of BSTs, operations are very quick.

Operation	BST
Search	$O(h)$
Insertion	$O(h)$
Deletion	$O(h)$
Max/Min	$O(h)$
Pred/Succ	$O(h)$

Table 4: BSTs

Note that the h in $O(h)$ refers to the tree's height. A perfectly balanced tree with an equal number of items on the left and right is shorter than an unbalanced tree. In a perfectly balanced tree, $h = log(n)$ where $n = number of items$.

- Search takes $O(h)$ time because we search through the tree by making a choice at each branch. On a perfectly balanced tree, we essentially perform binary search.
- Insert takes $O(h)$ time because we just go all the way down the height of the tree.
- Deletion takes $O(h)$ time to fix the BST after deleting the item.
- Max/Min take $O(h)$ time because we just go the the left-most item for min and the right-most item for max.
- Transversal takes linear time.

There are implementations of BSTs that will maintain the tree to be as close to a perfectly balanced tree after any operation. These have a slightly larger overhead on operations.

As for implementations, most of the operations are fairly simple. Deletion is the tricky one. Deleting an item from a BST means that there are a few different cases:

1. Deleting an item with no children is simple.
2. Deleting an item with only one children is still simple.
3. Deleting an item with two children is tricky. In this case, the correct method is to find the min of the right sub-tree of the parent that will be deleted.

Priority Queues

Priority queues are data structures that work based on the value in a key, value pair. The idea is that while a regular queue is FIFO, a priority queue only cares about your priority.

Items are processed based on their priority instead of when they were inserted. Priority queues can be found in a lot of places. Waiting times in a clinic may be a form of priority queues.

A priority queue has a few basic operations:

- Insert.
- Search max/min.
- Delete max/min.

There are a few different ways to implement a priority queue.

Operation	Unsorted Array	Sorted Array	Balanced BST
Insertion	$O(1)$	$O(n)$	$O(h)$
Search max/min	$O(n)$	$O(1)$	$O(h)$
Deletion max/min	$O(n)$	$O(n)$	$O(h)$

Table 5: BSTs

For an unsorted array:

- Insert is constant.
- Searching for max/min is linear unless you keep a pointer to the minimum or maximum element.
- Deleting max/min is linear because you need to search for a new min/max.

For a sorted array:

- Insert in linear.
- Searching for max/min is constant if you know the size of the array or have a pointer to the max element. Searching for min is always constant.
- Deleting min/max is constant as long as you do not have to fix holes.

For a balanced tree:

- Insert is logarithmic because we just need to go down the balanced tree.
- Searching for max/min is constant.
- Deleting max/min is logarithmic because we fix the tree down.

Hash Functions

The **hash function** is a mathematical function that outputs an identifier for an input. Complex hash functions like the md5 create unique outputs for any input. All hash functions are math functions, i.e. a specific input will always have the same output.

Hash Tables

Hashtables are a data structure that allow very quick searching and updating. Updating compelxity depends on implementation.

Hashtables use a hash function to get an identifier or key for any item to store. This key may be reduced or lowered by using the modulus function. The final key can be thought of as a specific bucket number in a large row of buckets.

An example might be student IDs. Our very simple hashfunction can be a function that returns the last two digits of an ID, i.e. $f(12345) = 45$. But this means we need 100 buckets because the last two digits of an ID go from 0 to 99. To further reduce the number of digits, we can use the modulus operation. If we do $45 mod 10 = 5$, then we could only get numbers from 0 to $10 - 1 = 9$.

But we must have more than 10 students. What happens if two students end up with the same key?

To solve this issue, we use **chaining**. Each bucket has a list (or array, etc.) of items that contains all the items that got matched to that bucket. If two student do end up with the same key, they will both be inserted into the list under that bucket.

Note that when two different items get matched to the same bucket, we call it a **collision**.

Other Uses

Hash functions are very useful in many different areas apart from just creating hashtables.

They can be applied to most situations that require comparing large pieces of data. The hash function can be used to get a unique hash code of the item we are searching for. Then, it is compared to the hash codes of the items that we are searching.

A specific example of hashing is checking file integrity. Let's say a company wants to ensure that a file sent in an email was not tampered with, they can give the file and its hash code. Then you could calculate the hash code yourself and compare it to what it should be. This is the basis for asymmetric cryptography.

Sorting

Sorting may be the most studied problem in computer science. There are many algorithms to sort because computers spend a lot of time just sorting things.

Sorting algorithms are often the basis for other, more complicated algorithms. Different sorting algorithms have different strengths and are designed for different tasks.

Sorting can be a precursor to many different tasks:

- Sorting can be used as a step to searching. Binary search can be applied after sorting to yield an $nlog(n)$ for searching an unsorted data struct.
- Sorting can be used to get item frequency in a list.
- Neighbouring items can be compared for uniqueness. It can be used to get uniqueness at the same time.
- We can check if two strings are anagrams by sorting them and comparing them.

There are many more applications that sorting can make efficient.

Selection Sort

The gist of selection-sort is simple. We run through all the items in an array left to right. As we go along, we find the minimum of the items to the right of our current item. If this minimum is less than our current item, then we swap the current item and the minimum.

All we're really doing is finding minimum after minimum. That's why naive selection-sort runs in $O(n^2)$. It finds the minimum linearly. selection-sort is also an in-place sorting algorithm.

If we could speed up finding minimum, we could increase selection-sort dramatically. Luckily, there are data structures that allow you to quickly get the minimum like heaps. This is discussed in the heapsort section.

Insertion Sort

Insertion sort is also fairly simple. We run through the entire array to sort. As we go along, we are dividing the array into two: sorted and untouched. The divider between the two is just the current item. Everything to the left of current item is sorted and to the right is untouched.

The sorted section is sorted in this method: Each iteration, we grab the current item and continuously swap it with the item to the left until it is in the sorted position.

Eventually, everything is sorted.

Insertion sort runs in $O(n^2)$ and is an in-place sorting algorithm.

Heapsort

To understand heapsort we have to understand selection-sort first. This is because heapsort is just a sped-up selection-sort.

Heaps

The trick to speeding up selection-sort is noticing that it takes linear time to find the minimum in each iteration. If we could utilize other data structures that allow quickly finding the min, we could speed up selection-sort dramatically.

We've already discussed binary search trees which we could use to solve this problem since they allow finding min/max in $O(h)$ time. Similarly, we could use a priority queue based on a binary search tree.

But we can also use something called heaps.

Heaps are trees but they're not quite binary search trees. They have no order apart from one rule. The parent must be a greater value than its children in a **max-heap** and the parent must be smaller than its children in a **min-heap**.

This means that finding an item in the tree will take linear time. Unless we're finding the min in a min-heap or max in a max-heap, then it will be constant time.

Heaps are also created without pointers, unlike BSTs. We can represent a heap in an array. We can do this because we can find the parent using a formula since the way a heap fills up is entirely predictable.

We know that a heap has $\leq 2^{levels}$ items because it's a binary tree. We know that the left child of an item at k is at $2k$ and its right child is at $2k + 1$. And the parent of a node n is at $floor(n/2)$. This can be a little tricky to visualize so work it out on paper.

When loading a heap, we always fill the left child before the right child. This means that the last level of a heap will probably not be entirely filled. If we were to represent this last level entirely, then it would be a large drain on memory for large heaps. Instead, if we just implement some checks and store the heap size, we only have to have n spaces in memory and no empty spots.

Now for some basic operations on heaps:

- Search.
- Insert.
- Extract min/max.

Search occurs in $O(n)$, as previously discussed.

Insert is fairly simple. We insert a new item at the end of our array that represents the heap. But then we need to ensure that this new item has no conflicts with its parent. If there is a conflict,

we swap the parent and the newly inserted item. Now we need to call this check&swap function recursively on its new position to ensure there is no conflict again.

Extracting the **min/max** is a little tricky. After we extract the min, we need to find the new min as well. To do this we need to put in a new temporary root and then fix the tree going down. We can grab the last node in the tree as the new root for convenience. Then we need to check and fix the tree going down.

To fix the tree going down, we need to remember that in the trio of a parent and its two children, the parent has a greater/lesser value than its children. And this means that if there is a conflict with the parent, we just need to swap the parent with the left or right child depending on the values. But we may not be done yet. Similarly to insertion, we need to call this fix function recursively with the old, now-swapped parent in its new position.

That is essentially the algorithm. We start at position 0, i.e. the root, after extracting the min/max and check if the children are lower/higher. If they are not we make the lowest/greatest child the new parent and call the fix_down function on the new location of the misplaced item. This process exits when there are no conflicts with the misplaced item or we get to the bottom of the heap. Note that this process takes $log(h)$ time because it goes down the height of the tree.

Lastly, note that heaps are very useful. The details discussed above can be very easily used to implement a priority queue or other structures.

Sorting

Heapsort is using heaps to quickly extract the min/max. As we've already discussed, extracting the min/max is constant but we need to fix the heap down so it's a $log(n)$ operation overall.

In an array of n items, we would extract the min/max n times so the total complexity of sorting an array would be $nlog(n)$.

Mergesort

Mergesort is based on one of the most powerful ideas in computer science, divide-and-conquer.

Divide and Conquer

The idea behind divide-and-conquer is fairly simple yet very powerful.

Generally, smaller problems are easier to solve than larger problems. This is the idea behind divide-and-conquer. We divide up a problem into many sub-parts and then solve each one of them.

Once the sub-parts are solved, we combine them to get a complete solved solution.

Sorting with Divide-and-Conquer

Mergesort uses divide-and-conquer to sort.

The idea is to recursively divide a array into many smaller arrays. Then we sort the small portions. After the small portions are sorted, we combine them back into a larger list or array.

The simplest implemention involves dividing up an array into halves recursively until you're left with just one item. Then, merge all these arrays with one item back together in the sorted order.

It's a little tricky to get a mental model of the recursion in mergesort. Just understand that by calling mergesort on the left half recursively and the right half recursively, it will work out. I find it difficult to visualize this but it is easy to walk through the steps to see why it works.

Visualizing Mergesort

```
def mergesort(l):
    if len(l) <= 1:
        return l

    middle = len(l) // 2

    left = mergesort(l[:middle])
    right = mergesort(l[middle:])
    full = merge(left, right)
```

Mergesort in Python3

Nevertheless, here is an attempt to visualize it. It's easy to see that by calling mergesort recursively on the left half, we'll eventually be down to one item.

Then we'll return that one item which brings us into the scope with an array with two items. We call merge sort on the right half of the array with two items returning just the 2nd item in the array. Now we merge the two in sorted order. We have a sorted array of two items at this point.

Now we'll go back from this scope to the one with an array of four items. We've already sorted the left but we'll call merge sort on the right which will do the whole business of calling itself until there's only one item and eventually you'll get another sorted array of two items.

Now we've got two arrays of two items that we'll merge together to get a sorted array of four items. This is going to continue until we end up merging the two halves of the initial, complete array yielding a final completely sorted array.

Merging Arrays

The trickiest part of mergesort is the merging. It helps to remember that the merge step receives two arrays. All the merge step needs to do is grab items from each array in increasing/decreasing order.

This is a simple problem. We know that each individual array is sorted already so we just need to check if the first item from array_one is less/greater than the first item from array_two. If yes, then grab the item from array_one and check again until both arrays are empty.

When both arrays are empty you will have a new sorted array.

Complexity

As for the complexity of merge sort, we only take $O(log(n))$ steps to divide and it's linear to sort the divided arrays. The overall complexity of merge sort is $O(nlog(n))$.

Quicksort

Quicksort is generally considered to be the fastest sorting algorithm. It runs in $O(nlog(n))$.

The idea behind quicksort is similar to divide-and-conquer as well. We take an array and choose a pivot point. The entire array is then divided into two sections: less than pivot, and greater than pivot.

Now we have three total sections. `less_than_pivot < pivot < greater_than_pivot`

The great thing is that pivot is now in its final position in the array. Now we just need to call quicksort recursively on the two other sections of the array until all items are in their appropriate position.

This recursion is easier to visualize than the mergesort recursion. That one is far trickier to visualize.

The tricky part of quicksort is the partition algorithm, i.e. the part that separarates the arrays into two parts.

Partitioning

The general idea of the partition function is as follows. The partition function takes in a pivot index and it returns the new pivot index after partitioning all the data.

To partition the data, first it will swap the initial pivot point and the first item in the array. This makes things a little neater. The next part gets a little tricky.

The goal is to separate the array into two sections: less than pivot and more than pivot. We do this by running through the array. But we keep a left hand and right hand pointers, i.e. just index trackers.

Initially, both these pointers point to the index one, i.e. right after pivot. Now when running through the entire array, we increase the right hand every time but the left hand increases only for a special case.

We only increase the left hand if the right hand is pointing to an item that is less than the pivot's value (assuming ascending order). And we also swap the items at left hand and right hand in this case.

The result is that after transversing the entire array, the only thing left is to swap the pivot with left_hand-1. The partition is now done.

The partitioning can be a little tricky to understand in the details above because they are describing a `C` program. It's a lot simpler to do quicksort with list comprehensions and other higher order constructs but they're likely slower.

Randomization

Quicksort's complexity is a little tricky to understand. This is because its speed depends on how efficient partition is. If partition always divides the array fairly equally, the algorithm runs quickly. But if partition divides the array into one item and the rest of the array, then quicksort will be very slow.

This variation in complexity is why the notion of worst case, best case, and average case exists. The average case for quicksort is likely $O(nlog(n))$ because we don't need the pivot to be perfectly in the middle every time. It still works by being close enough.

The selection of the pivot point is what controls the speed of quicksort. One way to select it is to just pick the middle of the array. Another is to select the first item or the last item.

If the pivot is always the first item or the last item and the array passed in is sorted or almost sorted, then quicksort will be very slow because partition will never partition efficiently.

A solution to this is to randomize the input array before processing. This always ensure that sorted data cannot slow down quicksort.

Another solution is to **random**ly select the pivot point at each step. Randomization can be and is used a lot for cases like selecting the pivot point. Randomized data has a tendency to usually be the average case which helps make our quicksort consistent.

If we select our pivot point randomly and randomize the input data, we can expect $O(nlog(n))$ performace most of the time. It will also be a little faster than other sorting algorithms.

Bucketsort

Bucketsort is similar to the buckets analogy in hashtables.

Bucketsort uses the idea of divide-and-conquer but in a slightly different way than normal. Bucketsort utilizes more space than most sorting algorithms.

The idea is that before sorting we just place items into ordered buckets. The buckets are known so it takes $O(n)$ to iterate through the entire list and place in buckets. concatanate all buckets together.

The complexity of bucketsort is a tricky to define. The reason for this is that if the distribution among the buckets is sparse, then we just waste time bucketing items.

Bucketsort works fastest if there are lots of buckets and a fairly even distribution. This is because we get lots of already ordered buckets with only a few items to actually sort.

Intricacies

There are a number of details and choices that we must be aware of when talking about sorting algorithms.

Choices

Most sorting algorithms allow you to decide whether to sort in ascending or descending order. This is usually done with the help of a comparison that the user can specify.

For key, value pairs, sometimes you may want the sort the keys only. Maybe you want to sort the values based on the keys or other combinations. All these can be implemented into an algorithm.

Special Cases

Apart from choices, sometimes you need to be aware of special cases. One special case is the equal case. What to do with items that have equal value?

Another problem is specific instances of a general problem in sorting. For instance, strings are usually repressented as a list. They could be sorted in a way very similar to lists but you need to decide to compare ascii values for letters or implementing an alphabet heriarchy somehow.

Graphs

Graphs are an incredibly useful concept in computer science. They can be found almost everywhere.

A graph is just a representation of a connected system. In a graph, there are **vertices** and there are **edges** connecting the vertices. A node can be anything. It could be intersections on a map, possible moves in chess, etc. The edges represent a relation by connecting two vertices.

Graphs can be used to model a wide variety of problems.

Generally speaking, a graph is represented with two items: a list of **vertices** and a list of **edges**. Edges are just a tuple containing two vertices.

Types and Terminology

Because graphs can be used to model a wide variety of situations, they also come in many different types for more accurate modeling.

A **directed graph** has one-way edges, i.e. the relation is a one-way relation much like some roads are one-way. An **undirected graph** has two-way edges. Relations work both ways. Note that directed graphs can also have two-way edges but they need to be explicitly specified. But undirected graphs cannot have one-way edges.

A **weighted graph** assigns a value to each edge. This means there is a cost to each relation. An **unweighted graph** has no cost or value assigned to any edge. Note that sometimes vertices are given weights rather than the edges but we can just calculate the edge weight using the node weights.

A **non-simple graph** contains edges that contain only one vertex called a **self-loop**. They can also contain edges that occur more than once. These are called **multiedges**. A **simple graph** contains neither.

A **sparse graph** contains relatively fewer edges than are theoretically possible. A **dense graph** contains far more edges. There is no formal distinction between a sparse and dense graph.

A **cyclic graph** contains cycles whereas a **acyclic graph** does not contain cycles. Directed acyclic graphs are called **DAG**s.

An **embedded graph** contains geometric information that is useful to the graph. An example of this is a geographic map. A **topological graph** contains no geometric relations. The absolute position of the vertices do not matter.

An **implicit graph** is built as you use them. **Explicit graphs** are build explicitly.

A **labeled graph** contains unique identifiers for vertices. An **unlabeled graph** does not.

The **degree** of a vertex is the number of edges that connect to it.

Searching a Graph

Searching and transversing a graph is the most useful function of a graph.

For the purpose of discussion, assume that a graph is represented with a dictionary that uses a vertex label as a key and a list of neighbours from that vertex as its values. Note that the list of neighbours is usually called an **adjacency list**.

Searching through this graph is relatively similar for the two primary methods: breadth-first search and depth-first search.

Breadth-First Search

Breadth-first search focuses on searching widely first. The idea is best illustrated with an example. Let's say you lost your remote in the living room and now you need to search for it.

There is an implicit, embedded graph here. The furniture, carpet, walls, etc. are all vertices and any relations between them are the edges.

We could search for the remote by looking at the closest places first. Is it on the floor in front of you? Is it right behind you? Are you sitting on it?

If this doesn't work, we can look a little farther. Let's check the sofa cushions. We can check under the table and sofa.

If we still can't find it, let's look even farther away. Check the other rooms in the house. Hopefully, by now you've found the remote.

Searching for that remote was an example of breadth-first search. We looked in the vertices closest to us first and expanded our search as if it was a growing circle.

Depth-First Search

We can use the same example to illustrate depth-first search. Let's lose our remote again.

This time, we check the sofa to the left for the remote. Then we check the floor behind the sofa. But it's not there so we go into the room to the left and check the floor there.

It doesn't seem to be there so we come back to the sofa and search the table in front of us. Then we go to the room in front of us. And we find the remote in that room.

This was an example of depth-first search. Note how our search for focused on following a trail deeper and deeper rather than checking our immediate surroundings first.

Implementation

Both of these searches are very similar in their implementation. They only have one difference between them.

Breadth-First Search First let's discuss breadth-first search.

Let's grab that graph that stores its data in a dictionary as described in the "Searching a Graph" section.

We need a starting vertex called a root. We'll also keep track of which vertex we've already been to. Sometimes, we also keep track of where the vertex we arrived at another vertex from.

Now all we do is get the neighbors of our root and add them all to a queue. Then we grab an item off the queue and check if we've visited it before. If not, then get its neighbors and add them to the queue. If we have visited it before, we just throw it out.

We continue this process until our queue is empty at which point all the neighbours of all the vertices, i.e. all vertices, have been processed.

To understand why this is breadth-first search picture a simple graph, and use the algorithm on it. You'll notice that the queue (FIFO) is the key here. The queue allows you to expand your search outward like an circle with an increasing radius.

Depth-First Search That queue is also the key to understanding the difference between breadth-first search and depth-first search.

The implementation for depth-first search is identical apart from one detail. Instead of using a queue, we use a stack. So now instead of FIFO we have LIFO. This makes the search go deep following trails as far as they go.

Note that although our algorithm would stop when the queue or stack is empty, that does not necessarily mean that all the vertices in a graph have been processed. But we know that all the connected vertices in that section of the graph have been processed.

More Terminology

All edges used in depth-first search can be classified into two categories: **back-edge** or **tree-edge**. Back-edges point back higher up into the graph and tree-edges point lower to unexplored vertices.

Other edges are **forward-edges**. These are like back-edges but they point forward instead of back into the graph.

A **strongly connect comonent** is a sub-graph that contains paths between a group of vertices.

Paths and Walks

Paths and walks are just an extension of searching graphs.

A **walk** is just a sequence of vertices that are connected. A **path** is a walk that contains no repeating vertices.

To find a path between to vertices, we can use the search algorithms. The only change we make is that when we get the neighbors of a vertex, we record which vertex we got there from.

This way, when we finally get to our end vertex, we can actually trace the vertices we travelled to arrive to the end.

Minimum Spanning Trees

So far, everything we have talked about has been for unweighted graphs. Weighted graphs have slightly different properties and allow different operations.

One operation is a minimum spanning tree. This is a tree containing a portion of all edges where the sum of these edges is a minimum.

There are a few algorithms to determine the minimum spanning tree in a weighted graph but we'll discuss Prim's algorithm.

Note that in an unweighted graph, there can be many MSTs. A valid MST is one that contains all the vertices with the fewest number of edges.

Prim's Algorithm

Prim's algorithm is a surprisingly simple algorithm that can be fairly quick depending on the data structure used.

It's not very difficult to understand either.

First, we pick a starting node and add it to a list of new vertices. Now we will find the cheapest edge from our starting node. This edge will be added to our list of new edges and the vertex that it connects to will be added to our list of new vertices.

Now we search for the cheapest edge that connects to either vertex in our list of new vertices. And we add the new vertex and its edge to our lists.

Now repeat the last paragraph until there are no more edges/vertices left. This will construct a minimum spanning tree.

As for performance, it depends on the data structure used. This most expensive part is finding the minimum edge that starts from a vertex in our list of new vertices.

If a priority queue was used to keep the edges from the list of new vertices, we could very quick obtain the minimum edge.

There are more ways to increase performance. The description of Dijkstra's algorithm below is very close to a quick implementation of Prim's algorithm.

Kruskal's Algorithm

Although no details will be included, it's worth mentioning another algorithm for MSTs.

Kurstal's algorithm is another algorithm that can find MSTs. Kruskal's algorithm finds the MST by using a priority queue of the cheapest edges. It treats all vertices as if they're their own connected components.

It takes the cheapest edge, checks if it's part of two separate components, and if it is, then the two vertices in the edge get merged into a single connected component.

This is done until the priority queue of edges is empty.

The performance of Kruskal's algorithm depends on how long it takes to figure out if two vertices are part of the same connected component.

There is a data structure called the **union-find** data structure that can help reduce the time it takes to check if two vertices are part of the same component.

Shortest Path

A very common problem with graphs is finding the shortest path between two vertices. This can be found everywhere in real life too. A great example is finding the quickest route between two points on a map.

Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm that helps find the shortest path between two vertices.

The way it works is by finding the cost to visit all vertices from a starting vertex. But it does this in a way that only the minimum cost is ever considered.

An implementation is also fairly simple. First, we pick a starting vertex. Now we will maintain a priority queue that we'll be throwing a bunch of vertices in.

After we've initialized, we grab all the neighbors of our starting point and throw them in the priority queue. The values in the priority queue are based on the cost to get to that neighbour from the starting point. This cost could be based on any comparison. An example might be geometric distance. We also keep track of each node's total cost from start and which node we took to get to the neighbour which in this case was the starting point.

Now we repeat but grab the neighbour with the lowest cost from the priority queue. Next, we add its neighbours to the priority queue. We repeat this process until we've found the destination node or our priority queue is empty, i.e. destination node is not in connected component.

To actually construct the path, we just need to follow the vertex that we arrived at the destination node from all the way to the starting point.

Complexity The performance of dijkstra's algorithm depends largely on finding the neighbour with the minimum cost. A priority queue helps keep this cost to a minimum.

Another way to improve performance is only finding the min of the neighbours you encounter. This is the way it worked in the description above. An alternative approach is to find the min among all vertices every iteration. That is slow and unnecessary.

Note that Dijkstra's algorithm is very similar to Prim's algorithm. Infact, small changes to the above implementation would turn it into Prim's algorithm.

Searching

Binary Search

Binary search is one of the most amazing algorithms in computer science. It is the iconic divide-and-conquer algorithm.

It's also quite simple to understand. Given a sorted list, we can identify a range where our item might be. At first the range extends to the whole array. But if we check the middle of the array, we can rule out half of the entire range.

For example, if the middle item is `5` and we're looking for `2` then we know that the useful range is the first half. The latter half cannot contain `2` because it is sorted.

Now we do this process recursively, always dividing our range into halves until we cannot anymore. At this point we either find our item or the item is not in the array.

Linear Search

Linear search might be the simplest algorithm in computer science. We just check every item in a set if it is the item we are searching for.

Backtracking

In a lot of problems, data won't be sorted and there will be a lot of data. Sometimes, you may only have implicit data. In cases like these, linear search and binary search are out of the question.

But we can use something called **combinatorial searching** to help find the solution. These are similar to brute-force but a little more clever. They should still only be used when no other solution exists. Backtracking is a form of combinatorial search.

The idea of backtracking is fairly simple. Instead of computing all the possible solutions to a problem and then using the optimal solution, we can build up the solution one step at a time eliminating lots of possible dead-ends along the way.

Backtracking is very similar to depth-first search. Infact, backtracking works on implicit solution graphs.

To use backtracking, the solution must be possible to find in a number of steps. These steps can be of any type. An example might be finding all permutations of a string. This is usually done in steps: picking the first letter, then another, etc.

Visualizing

The general backtracking algorithm starts with an empty solution. It will then create a list of all possible steps onwards from the current state.

The function that creates all the new possible steps holds a lot of power. It can implement many pruning methods and fill in the new moves in a smart way.

Pruning is a technique used in backtracking to quickly rule out some solutions saving time. If you find that after adding a step, the solution is definitely wrong then we can prune that sub-tree and move on saving valuable time.

It will then add the new step to our solution and check if it is the solution we are looking for. If it's not that it'll continue adding steps until it cannot anymore.

When it's stuck and can't add any more steps, it will start to remove steps until it can add another new step. Then it will go down that path like before.

Sudoku

The sudoku example in The Algorithm Design Manual may be the best example for illustrating backtracking and efficient pruning.

The gist of it is as follows. To solve a sudoku puzzle, we will fill in solutions one by one until every item is filled.

But the hard part is figuring out which space to fill and what possible values can go in that space. The example illustrates that even simple pruning drastically reduces running time.

Smart pruning, by looking ahead in a solution for possible values, dramatically reduces the algorithm to pretty much instantaneous.

The takeaway is that even if there is no perfect algorithm to solve a problem, we can usually do much better than simple brute force.

Simulated Annealing

Combinatorial search finds the optimal solution faster than brute-force but it is still going to be very slow on extremely large data sets.

For these situations, we can use **heuristic searching**. We have no guarantee of finding the optimal solution but we can try to get as close as possible. Simulated annealing is a form of heurstic search.

The basic idea is as following. If we think of solutions as a graph (calculus), then there are local minimums and maximums. But ideally, we want the global maximum, i.e. the best solution.

One way to solve this is with a local search. This searches nearby the initial solution for a better solution. The problem with this approach is getting stuck in local maximums or minimums.

To solve this, use simulated annealing. Simulated annealing borrows ideas from actual annealing in metallurgy. When the temperature of metal is high, atoms generally lose energy causing temperature to drop. But the temperature is the average energy. Some atoms can jump to a higher energy as well. But the probablity of an atom to jump to a higher energy level is lower at lower temperatures.

Simulated annealing works in a similar way. We choose an initial solution and temperature. Now we start looping while looking for other solutions randomly nearby.

If this solution is better than one current solution, we definitely move to it. If it's worse then it's tricky. We might move to a worse solution if the temperature is high enough and the solution isn't too much worse.

And every n iterations, we decrease the temperature. The temperature is decreased exponentially.

What happens is that while the temperature is high, the algorithm will be trying out all sorts of solutions even if they're a little worse hoping to find the global maximum elsewhere. As the temperature slows, it'll become more conservative with taking risks on worse solutions.

The algorithm quits when we've found a satifactory solution or after a certain number of iterations.

This heuristic is very good at solving difficult problems without algorithms. It can be used on the travelling salesman problem to get a solution very close to the best special-purpose algorithms for that problem.

The actual implementation is complicated and not mentioned here. Just remember that this option is available for problems.

Random Sampling

Random sampling is another form of heuristic searching. This is best for cases where there doesn't appear to be a way to logically come to a solution.

The best bet in those cases is just to randomly select a sample and check if it can be a solution.

Dynamic Programming

At it's core, dynamic programming is about improving an algorithm by storing partial results to avoid recomputing.

Memoization

Memoization is a key dynamic programming concept. It's best to use to the canonical example to illustrate memoization.

If we were to try and calculate the fibonacci sequence for some number n.

$$fib(n) = fib(n-1) + fib(n-2) \tag{11}$$

Note how the fibonacci sequence is recursive. This means that there's overlap between the recursions. To see this, look at the flow in calculating $fib(6)$.

$$fib(6) = fib(5) + fib(4) \tag{12}$$

$$fib(5) = fib(4) + fib(3) \tag{13}$$

$$@fib(4) = fib(3) + fib(2) \tag{14}$$

Note how we've already had to compute $fib(4)$ twice? Once when calculating $fib(6)$ and once in calculating $fib(5)$.

Memoization solves this problem by caching the results of computations that we may use later. In this case, we can store every fibonacci value we calculate.

This allows us to check if we've already calculated a fibonacci number and then retrieve the value in constant time if possible. This approach completely removes any repeated computations.

The impact of memoization of fibonacci numbers is enormous. We bring the complexity from $O(2^n)$ down to $O(n)$.

This was possible from the tradeoff of repeated computation with using more space. This is the downside of memoization.

Note that in this specific example, if we were really clever, we'd notice that the fibonacci sequence only ever needs to store the last two fibonacci numbers. If we do exactly that, we can still calculate the fibonacci sequence in linear time while using minimal space.

Longest Common Subsequence

The problem of finding the longest common subsequence is another great example of memoizing and dynamic programming dramatically reducing the complexity.

First, the algorithm. I've had more trouble with this algorithm than most others. The explanation may be a bit verbose.

Let's say we have two strings, A and B. Remember how strings are recursive structures? We can use that property now. Let's say we decide to compare the last character in the strings.

There are only two cases: they match or they don't match. If they match, then we know that whatever the LCS (longest common subsequence) is, it must contain this character. So now let's just cut this character off both A and B and find the LCS of the smaller strings. We know that the overall length of the LCS must be `1 + LCS(A[:-1] + B[:-1])`.

What about the other case? If they don't match, then there are two possibilities. We know that both `A[-1]` and `B[-1]` can't be in the LCS. But that still leaves the case where neither or one of them is in the LCS. For those cases, we can call `max(LCS(A[:-1] + B), LCS(A + B[:-1]))`

This would take care of the three cases when the characters do not match. Now we just need an end condition. That's simple. When either string has a length of zero, we know we cannot go any further and the LCS of the strings must be zero.

This algorithm will give the length of the longest subsequence. To visualize it, let's look at a simple example.

Example

Let's find the LCS of `Boo` and `Look`.

We check the last characters, `o` and `k` are not the same so now we must call LCS on the shorter strings `Bo, Look` and `Boo, Loo`.

For `Bo, Look`, the last character does not match again so we call LCS for `B, Look` and `Bo, Loo`.

On the other side, we have `Boo, Loo` which we use to find `Bo, Loo` and `Boo, Lo`.

This is getting tedious but note that we've already have to find `Bo, Loo` in two different places.

This already shows an opportunity for dynamic programming. If we could keep a dictionary that contains the LCS values and pass it between calls, we could save a lot of time. We can do exactly this by storing LCS values using the length of the two strings.

If we were to not memoize anything, the algorithm would be exponential but if we do memoize partial results, we can get it down to just $O(mn)$ where $m = length(A)$ and $n = length(B)$.

Strings

The algorithm we talked about above will return the length of the longest common subsequence but it won't actually tell us what that subsequence is.

We can build another function that uses the memo filled by `lcs` to return the string.

The idea is simple, picture the memo storing LCS values in a dict using the lengths of two strings. This memo can be visualized as a 2D table. The rows and columns are made up of letters from the two strings.

Our goal is to start off at the bottom right and end up at the top left. The path we take is decided by the following rules:

If the last character in the strings match, we print that character and call the function recursively on the strings minus that last character.

If the last characters do not match, we compare the values above and to the left of our current value in our 2D table. We follow whichever side has the greater LCS value.

Note that our current value is just the lengths of our two strings, i.e. `memo[(len(s1), len(s2))]`.

Eventually, you will get to an empty string and return.

Longest Increasing Subsequence

A naive method to find the longest increasing subsequence is to sort a copy of the sequence. Then find the longest common subsequence between the raw sequence and the sorted sequence. This runs in $O(n^2)$.