Exploring advanced programming concepts

Sjors van Gelderen

January 25, 2017

Contents

1 Introduction

In this document I describe the materials studied during my graduation phase. The aim of the project was to prepare for professionally teaching students selective aspects of computer science. These aspects include:

- The C# and Python 3 programming languages
- Object-oriented design patterns
- Data structures, algorithms and complexity analysis

I have provided many example implementations in a variety of programming languages. All provided complexities in this document and related implementations are worst-case complexities.

1.1 Primary languages

Here is a brief description of the main languages used.

1.1.1 C#

C# is Microsoft's most popular .NET programming language. The language strikes me as a much higher level variant of C++. With its rich .NET ecosystem, modern features (LINQ, anonymous methods, tasks, etcetera), C# is certainly a powerful programming language.

Sources C# code is provided in the subdirectory *csharp/*.

1.1.2 Python 3

Advantages of this language include its concise syntax and its portability. Because of Python 3's high-level nature, the programmer can focus exclusively on the actual logic of an algorithm, rather than the low-level memory management involved. This is removes a significant source of potential distraction. Unfortunately, the absence of a strict compiler means frequent run-time debugging sessions are necessary.

Sources Python 3 code is provided in the subdirectory *python_3/*.

1.1.3 F#

Being a more recent addition to the .NET family of programming languages, F# has not quite gained the popularity of C#. The F# programming language enables the programmer to tackle problems using high-level concepts such as recursion, higher order functions, partial application and currying, computation expressions (syntactic sugar for monads) and more.

Sources F# code is provided in the subdirectory *bonus/fsharp/*.

1.2 Secondary languages

These languages were only used sparingly, out of curiosity rather than necessity.

1.2.1 C

Virtually any programmer will at some point in their career encounter this venerable, fast and portable programming language. Because this low-level language doesn't use a garbage collector, the programmer must exercise great caution with the manual management of memory. Many security problems that affect us today are a direct consequence of a failure to do so. This language is well-suited to studying the low-level implementation of algorithms and data structures.

Sources C code is provided in the subdirectory *bonus/c/*.

1.2.2 Rust

Developed by Mozilla, Rust aims to be a modern solution for safe, asynchronous programming. With default immutable variables, as well as borrowing and lifetimes, the compiler makes it very difficult indeed to write a program that contains run-time errors relating to incorrect memory access.

Sources Rust code is provided in the subdirectory *bonus/rust/*.

1.2.3 Chicken Scheme

The LISP family of programming languages has two major dialects; these being Common LISP and Scheme. Chicken Scheme is a modern implementation of the Scheme dialect. It has a very minimalistic syntax, revolving around the use of parentheses and prefix notation. Chicken Scheme is a functional programming language.

Sources Chicken Scheme code is provided in the subdirectory *bonus/chicken_scheme/*.

2 Analysis

2.1 Empirical analysis

Empirical analysis refers to inferring a program's expected performance based on measurements taken while running a program with various configurations. A typical scenario involves the measurement of performance in terms of running time by the use of a 'stopwatch'.

2.2 Complexity analysis

Complexity analysis is different from empirical analysis in that it uses reasoning - rather than experiment - to determine a program's expected performance.

3 Data structures

As the manner in which data is stored affects the efficiency of and compatibility with a given algorithm, it is appropriate to discuss the studied data structures now.

3.1 Array

Among the most common data structures for collections is the array. All elements in the array are stored contiguously in memory, and may be accessed in constant time($O(1)$) through their respective indices. Arrays have excellent cache-alignment, making them very fast indeed.

The location of an element in an array at some given index i will be

$$base\_address\_of\_array + i * s$$

In which s is the size of a single element. This size depends on the data type of the array.

3.1.1 Size

Fixed size By default, arrays are generally of *fixed size*; meaning the array will take up constant space (*O(1)*). Since less elements might be stored in the array than its total capacity allows for, space might be wasted on unused memory. The amount of elements that are actually being used is called the *logical size* of the array.

Dynamic size When an array is of dynamic size, the programmer must carefully specify the amount by which the array will be resized. If the programmer adds more elements than some given threshold will allow, the array must perform a resize operation. Since the resize operation is costly (typically $O(n)$), frequent resizes ought to be avoided. In some implementations, dynamic size arrays will also shrink when the logical size becomes less than a given threshold.

3.1.2 Dimensionality

Arrays may have more than one dimension. Such a construction may also be called a matrix. Elements in the multidimensional array may be accessed with multiple indices describing the relevant coordinates.

3.2 Linked list

The *linked list* is a linear, dynamic size data structure for storing collections of elements. Contrary to arrays, elements are not guaranteed to be stored contiguously. This makes it possible to store elements in a fragmented fashion, which is useful when the collection is larger than any available contiguous block of memory. Since elements in the linked list are stored in a fragmented fashion, direct access through indices as done with the array becomes impossible.

The linked list consists of several segments, each of which has up to two references to other elements in the sequence. The last element of the sequence will point to an empty segment, which may be called a *null link.* Traversing the linked list is a linear time ($O(n)$) operation.

Singly-linked and doubly-linked The *singly-linked* list consists of segments that contain a value and a reference to the next segment in the sequence. The *doubly-linked* list is the same as the singly-linked list, except each segment also has a reference to the previous segment in the sequence. Whether the list is singly-linked or doubly-linked affects the traversal process, since where singly-linked lists only allow forward traversal, doubly-linked lists also allow backward traversal.

Insert Elements are inserted at the root of the list. Since no traversal is required to locate the root, the insertion operation is one of constant time $O(1)$. The inserted element becomes the new root, and is given a reference to the next segment. This next segment is the old root segment.

When inserting at the end - rather than the beginning - of a list, the list must be traversed until a given segment references a null link as the next element in the sequence. The reference to the empty segment may be replaced with a reference the element to be inserted. Due to the traversal, the worst case complexity of such an insertion is of linear time $O(n)$. If the list is *doubly-linked*, the new element's *previous* variable should reference the last element of the current list.

Delete Locating the element to be deleted will in the worst case require a complete traversal of the list, which is a linear time ($O(n)$ operation. If the current segment's *next* variable references the element to be deleted, it must be changed to point to the segment in the sequence that succeeds the element to be deleted (which may be *null*). If the root element is deleted, the next segment in the list becomes the root.

3.3 Stack

The *stack* is a *LIFO* (Last In, First Out) data structure. Data is always inserted on and removed from the top of the stack. Only the top element of the stack may be inspected at any time. Regardless of which structure is used to store the stack contents, the interface must always offer the following operations: *push*, *pop* and *peek*.

Push The *push operation* puts a new element on top of the stack. This is a constant time ($O(1)$) operation. If the stack has a size limit, this operation throws a *stack overflow* exception when the stack's capacity is exceeded.

Pop The pop operation removes the top element from the stack. This is a constant time ($O(1)$) operation. If the stack has no elements, this operation will throw a *stack underflow* or *invalid operation* exception, as there are no elements to remove.

Peek The peek operation gives the programmer access to the current top element of the stack. This is a constant time ($O(1)$) operation. If the stack is empty, this operation will throw an *invalid operation* exception, as there are no elements on the stack to reveal.

Sources Stack implementations are provided at the following locations:

- *csharp/data_structures/stack/*
- *python_3/data_structures/*
- *bonus/c/data_structures/*
- *bonus/fsharp/data_structures/*

3.4 Queue

Contrary to the stack, the queue is a *FIFO* (First In, First Out) data structure. Elements are inserted at the back of the queue, and removed from the front of the queue.

Enqueue The *enqueue* operation adds a new element at the front of the queue. This is a constant time ($O(1)$) operation.

Dequeue The *dequeue* operation removes the element at the back of the queue. This is a constant time ($O(1)$) operation for arrays, and a linear time ($O(n)$) for linked-lists.

IsEmpty The *is_empty* operation returns whether the queue is empty or not. This is a constant time ($O(1)$) operation. It can be used to prevent queue underflow.

Rear and front The *rear* and *front* operations provide access to the front and rear elements of the queue. These operations are optional and as such are not provided in all implementations. Both operations are constant time ($O(1)$) operations for arrays. The rear operation is linear time ($O(n)$) and the front operation is constant time ($O(1)$) for linked-lists.

Sources Queue implementations are provided at the following locations:

- *csharp/data_structures/queue/*
- *python_3/data_structures/*
- *bonus/c/data_structures/*
- *bonus/fsharp/data_structures/*

3.5 Hashmap

This data structure - also known as a *dictionary* or *hash table* - is used to store a list of key-value pairs. This kind of structure is also called an *associative array.* The data structure consists of a list or array of *buckets* or *slots* containing one or more elements. The amount of elements that may be stored in a given bucket depends on the implementation of said bucket. The underlying data structure may, for example, be a fixed-size array or a linked-list.

Of primary interest is the way the indices of the relevant bucket are determined. Unlike standard arrays, indices need not be integers. Indices may instead be derived from any kind of data type. This is done using a *hash function* (hence, the name of the data structure).

The hash function derives an integer from the provided data. The modulo operator then restricts the provided integer to a range that fits in the array or list.

$$hash\_value = hash\_function(data) \tag{1}$$

$$hash\_map\_index = hash\_value \ \% \ hash\_map\_size \tag{2}$$

To determine whether the number of buckets is appropriate for the amount of elements to be inserted, the load factor may be calculated. The load factor is calculated as follows:

$$\frac{number\_of\_entries}{number\_of\_buckets} \tag{3}$$

A large load factor is indicative of an inappropriate amount of buckets. If the load factor is too small, however, it is possible a lot of buckets are unnecessarily taking up space. Some implementations of hash maps will resize when the load factor reaches a certain threshold. This is a costly operation, seeing as all elements will have to be redistributed according to the new size.

It is possible that two different pieces of data generate the same hash map index. Such a situation is called a *collision.* If there are many collisions, the distribution of the values will be poor. As a consequence, the data structure will be inefficient. If all keys are known before the data is inserted into the hash map, it is possible to generate a perfect hashing function. This means each piece of data provided will result in a unique hash map index.

There exist several solutions to the collision problem, three of which are described in the next sections.

3.5.1 Linear and quadratic probing

When performing *linear probing*, a collision causes the hash map index to be increased successively by one, until an empty bucket is found. Similarly, *quadratic probing* successively increases the hash map index using values from a quadratic polynomial to find a new hash map index for the colliding element. Usually, both of these probing methods will wrap around if the resulting hash map index is outside of the range of the bucket array. These methods may cause elements to occupy buckets that would otherwise be populated by other pieces of data. In addition, these probing methods may cause infinite loops when an acceptable bucket is never found.

3.5.2 Dynamic size buckets

Another solution to the collision problem is to allow multiple values per bucket. Such a solution could be implemented by making each bucket into a dynamic size array or linked-list. If hash function or load-factor is bad, this implementation may result in similar efficiency to simply using a single dynamic size array or linked list.

3.6 Tree

Trees are hierarchical data structures consisting of *linked nodes*. Unless the tree is empty, there will be a top node called the *root node*. From this node spring all the *subtrees*. Any node other than the root node of the tree is called a *child node*. The predecessor of a child node is called its *parent*. The root node is the only node that does not have a parent. Nodes that don't have any children are called *leaves*, expanding on the analogy of the tree. Any node that has at least one child is an *internal node*. A tree is said to be *balanced* if the nodes are distributed evenly among the subtrees in the tree. A tree might be so poorly balanced that the nodes are organized much like a *linked list*. In this case, it's possible that the tree was not a suitable data structure for the relevant problem. The *depth* of a node is the amount of *edges* that must be traversed from the root to reach it. A collection of edges to traverse is called a *path*, the length of which is the amount of edges. All nodes of the same depth are said to be on the same *level*. In this section are described a number of data structures that may be called trees.

3.6.1 Binary tree

The binary tree - also called *bifurcating arborescence* - consists of nodes containing at most one parent, and at most two children. There is no particular property governing where new elements are inserted. This is because the binary tree does not inherently imply any particular sorting order. A binary tree is said to be *full* if all of its leaves are on the same level, and each internal node has two children. This is different from the notion of completeness, as a *complete* binary tree is a tree where each level other than possibly the last is fully populated, and all nodes are as far to the left as possible.

Sources A binary tree implementation is provided at the following location:

- *bonus/fsharp/data_structures/*

3.6.2 Binary heap

This complete binary tree maintains the *heap property*, which is said to be *max* or *min*. A heap that satisfies the *max heap property* is called a *max heap*. Accordingly, a heap that satisfies the *min heap property* is called a *min heap*. The property determines the order in which the elements are stored inside the heap.

In a max heap each key must be greater than every key stored inside its children. Conversely, in a min heap each key must be less than every key stored inside its children.

Insert The *insert* operation adds a new key at the end of the heap. If the heap property is violated, it may be restored by letting the key 'swim' to the correct position. When 'swimming', the key is compared to its parent. If the a violation of the heap property is found, the key and its parent are swapped. This process is repeated until the heap property is no longer violated. Note that the heap property is also restored if this key becomes the new root of the heap.

Extract and heapify The *extract* operation removes the root node from the binary heap. It does this by replacing the root key with the last key in the heap. After this step, the *heapify* operation is used to restore the heap property if it is be violated by the new root. In such case, the root is compared to its children. Depending on the relevant property, the root is swapped with either its smaller child in a min-heap, or its larger child in a max-heap. The process is then repeated from the old root's new location, which is the index of the child it was swapped with. Eventually, the heap property must be restored.

Sources Heap implementations are provided at the following locations:

- *csharp/data_structures/heap/*
- *python_3/data_structures/*
- *bonus/fsharp/data_structures/* (At the time of writing incomplete)

3.6.3 Binary search tree

Similar to the binary heap, the binary search tree must satisfy an ordering principle: the binary search tree property. This property allows for more efficient searching algorithms, as the partitioning of the tree is known. The binary search tree property states that all keys in the children of the current node must be less than the key of the current node, and all keys in the right children of the current node must be greater than the key of the current node. In a standard implementation, duplicate values are generally not allowed.

Traversals Visiting elements in a tree is called *traversing* the tree. The most common traversals are the following:

Pre-order traversal Visit the root, visit the left subtree, visit the right subtree.

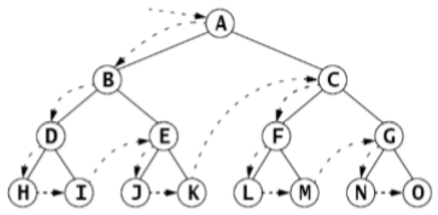

Figure 1: Pre-order traversal diagram

In-order traversal Visit the left subtree, visit the root, visit the right subtree.

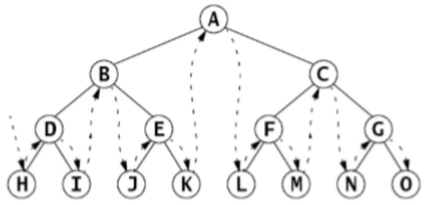

Figure 2: In-order traversal diagram

Post-order traversal Visit the left subtree, visit the right subtree, visit the root.

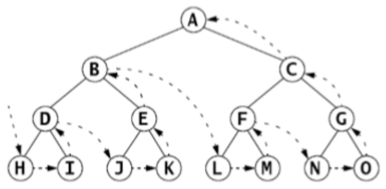

Figure 3: Post-order traversal diagram

Insert The *insert* operation is used to insert new keys into the tree. This is a linear time ($O(n)$) operation. Since the binary search tree property must be maintained, the insert operation has to intelligently locate the correct position for the new element. The insert operation always starts from the root of the tree, which may be empty. If the current node is not empty, the algorithm will continue on the left subtree if the value to be inserted is less than the value in the current key. Otherwise, it will continue on the right subtree. This process is repeated until an empty node is found. When it is found, the new element will populate the empty node.

Delete The *delete* operation is used to delete existing keys from the tree. This is a linear time ($O(n)$) operation. First, the element to be deleted must be located. Once the target key is found, the algorithm must establish how to proceed with the deletion. If the node to be deleted does not have any children, the delete operation is trivial, as the current node can simply be erased. If the node has a single child, that child will take the place of the node to be deleted. A more complex situation occurs if the node has two children, as the binary search tree property must be maintained. To do this, the algorithm must either locate the *in-order successor* or *in-order predecessor* of the target node. The located node's key then replaces the deletion target node's key. After this, the delete operation continues on the located node until one of the trivial cases occurs.

Search The *search* operation is a linear time ($O(n)$) operation that finds a target node based on its key. Since the binary search tree property holds, the search operation may from any key easily determine whether to branch the search operation to the left or right of said node. If an empty node is encountered, the key is not in the tree.

Sources Binary search tree implementations are provided at the following locations:

- *csharp/data_structures/binary_search_tree/*
- *python_3/data_structures/*
- *bonus/c/data_structures/*
- *bonus/fsharp/data_structures/*

3.6.4 K-dimensional tree

The *K-dimensional* tree expands upon the philosophy of the binary search tree, in that it must maintain a sorting order property. When using this tree in less than 4 dimensions, a simple geometric interpretation of its structure is possible. Beyond this, the analogy becomes more difficult to grasp as it involves the splitting of hyperplanes.

Insert The *insert* operation of a K-dimensional tree is of linear time ($O(n)$). Let's examine the construction of a *2-dimensional* tree from the array $A = [(3,2),(6,5),(7,1)]$. The root key of the tree will be the first element in the array, in this case $(3,2)$. We will record this key as a *vertical* split of the plane, like so:

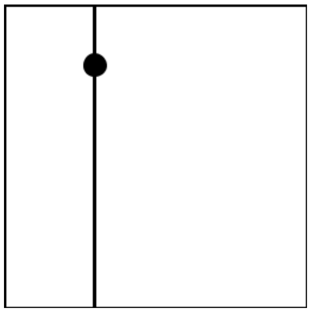

Figure 4: Node $(3,2)$ is splitting the plane vertically

Since the next key in the collection - $(6,5)$ - has a larger X coordinate than the root key($6 > 3$), it will be stored to the left of the root key. Notice that this operation has added a new level to our tree. In a 2-dimensional tree each level of the tree will be recorded as being either *vertical* or *horizontal*. For this example, let's assume that even levels are vertical, whereas odd levels are horizontal.

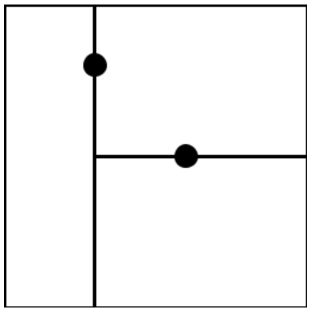

Figure 5: Node $(6,5)$ is splitting the subsection of the plane horizontally

The next key in our collection - $(7,1)$ - has a larger X coordinate than our root key($7 > 3$), and so the key must be stored in the right subtree. The right subtree is not empty, therefore we must compare the right child of the root key to the key we're inserting. Since the right child of the root key is on an odd level in our tree, it corresponds to a *horizontal* split. The consequence of this difference is that we must now compare the Y coordinates - rather than the X coordinates - of our keys. Since the Y coordinate of our insertion key is less than that of the right child of the root key, we must store our new element to the left of the right child of the root key.

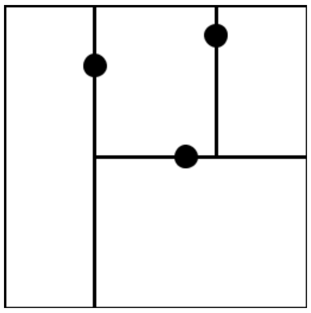

Figure 6: Node $(7, 1)$ is making yet another vertical split

Search The *search* operation for a key on a k-dimensional tree is of linear time ($O(n)$). When searching for a key, the orientation of each level must be taken into account. Similar to the insertion operation, the X or Y coordinates will be compared to determine whether the left or right subtree is to be used for the rest of the search operation. If they key is encountered, the search is succesful. If an empty tree is encountered instead, the search was not succesful as the key is not in the tree.

Range query The *range query* operation is of $O(\sqrt{n} + k)$ time complexity, where k denotes the number of reported points. In the 2-dimensional tree, this operation queries a rectangular section of the plane and returns the points contained within that section.

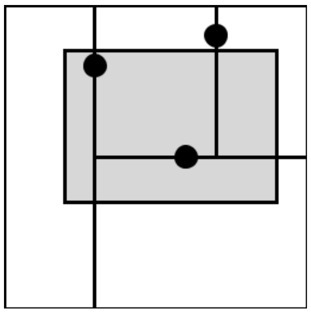

Figure 7: A range query is performed on the rectangular area $(2, 1.5, 9, 6.5)$

As with the operation that searches for a single key, the range query must take notice of the orientation associated with each level. Rather than always continuing the query on a single child tree, the range query may continue on both children if the section covers space on both sides of the current key's split.

Sources A K-dimensional search tree implementation is provided at the following location:

- *csharp/data_structures/kd_tree/*

3.7 Graph

The *graph* may be used to represent a relational network of nodes called *vertices*. Vertices are connected via *edges*. Vertices that are connected to each other via a single edge are called *adjacent* or *neighbors*. A collection of edges connecting two vertices is a *path*. The graph does not imply any particular root, nor does it restrict the number of edges connecting any vertex to other vertices.

3.7.1 Weighted

A graph is *weighted* if each of its edges has a 'cost' associated with traversing it. Such graphs may, for instance, be used to map distances between locations on a map.

3.7.2 Directed

A graph is *directed* if each of its edges has an associated direction. This direction implies a restriction on the traversal, as only the direction given for each edge is allowed to be travelled. A directed graph is also called a *digraph*.

3.7.3 Dense versus sparse

Graphs are called *dense* if there are many edges connecting many vertices. Conversely, a graph may be called *sparse* if there are comparatively little edges connecting the vertices.

3.7.4 Adjacency list

Graphs may be structured as an *adjacency list*. This is a list of vertices, each of which is associated with all of its neighbors.

A	B C D
B	A D
C	D A
D	C B A

A	B D
B	D
C	A
D	C

Figure 8: Examples of adjacency lists. The right list is for a digraph

The adjacency list is the preferred representation when the graph is sparse. In a digraph, only edges that *emanate* from (start at) the vertex are recorded for that vertex.

3.7.5 Adjacency matrix

The *adjacency matrix* is a multi-dimensional array. The axes both contain each vertex in the graph. If two vertices are adjacent, it is recorded in the matrix. The recorded value may be a boolean type, in which case the value is *true* if the vertices are adjacent. If the graph is weighted, the value may contain the weight of the connecting edge instead of the boolean value. In this case, the weight for non-adjacent vertices could be recorded as being infinite.

	A	B	C	D
A	F	T	T	T
B	T	F	F	T
C	T	F	F	T
D	T	T	T	F

	A	B	C	D
A	∞	T	T	T
B	T	∞	∞	T
C	T	∞	∞	T
D	T	T	T	∞

Figure 9: Examples of adjacency matrices. The right matrix is for a weighted graph

The adjacency matrix is the preferred representation when the graph is dense.

3.7.6 Incidence matrix

The *incidence matrix* shows the relationship between vertices (represented in the rows) and edges (represented in the columns) in a graph. If the vertices are incident upon (connected to) the edge, a boolean value or weight may be recorded in the matrix.

	0	1	2	3
0	F	T	T	T
1	T	F	F	T
2	T	F	F	T
3	T	T	T	F

	0	1	2	3
0	∞	T	T	T
1	T	∞	∞	T
2	T	∞	∞	T
3	T	T	T	∞

Figure 10: Examples of incidence matrices. The right matrix is for a weighted digraph

In a digraph, edges that emanate from the row vertex are given the value 1. Edges that terminate at the row vertex are instead given the value -1.

3.7.7 Traversal

Here are listed two ways to visit each vertex in a graph, both of which are of linear time ($O(n)$) complexity.

Breadth-first search The *breadth-first search* uses the *queue* data structure. The general procedure is as follows:

1. Enqueue the root vertex
2. Dequeue a vertex. If this is the target, stop the search. Otherwise, enqueue all unvisited direct children

3. If the queue is empty, the traversal is complete and the search was unsuccesful. Otherwise repeat step 2

Depth-first search The *depth-first traversal* uses the *stack* data structure. The general procedure is as follows:

1. Start at the root vertex
2. If the current vertex is the target, stop the search. Otherwise, continue at step 3
3. Push the first unvisited neighboring vertex onto the stack
4. Repeat step 3 until no more unvisited neighbors exist, then pop from the stack. Repeat from step 2

4 Algorithms

4.1 Aspects of algorithms

Time and space complexity

Asynchronous / concurrent algorithms

Divide and conquer

Greedy algorithms

4.2 Searching

4.2.1 Linear search

The *linear search* traverses a collection in linear fashion until either the requested element is found, or the end of the collection was reached. This is a linear time ($O(n)$) operation. Linear search is simple to implement and is well-suited to small unsorted collections.

Sources Linear search implementations are provided at the following locations:

- *csharp/algorithms/searching/*
- *python_3/algorithms/*
- *bonus/c/algorithms/*
- *bonus/fsharp/algorithms/*
- *bonus/rust/searching/*
- *bonus/chicken_scheme/algorithms/*

4.2.2 Binary search

If the collection is known to be sorted, the *binary search* algorithm may yield a significant performance gain compared to regular insertion sort. This is because the algorithm can use a pivot to intelligently search the collection. This is a logarithmic time ($O(logn)$) operation.

4.2.3 Peak finding

4.2.4 Hill climbing

Sources

- *csharp/algorithms/searching/*
- *python_3/algorithms/*
- *bonus/rust/algorithms/searching/*

4.3 Sorting

Here are described various sorting algorithms of which I have provided implementations in various programming languages.

Aspects of sorting algorithms

Stability

Comparison sort

4.3.1 Insertion sort

Perhaps the most basic sorting algorithm is *insertion sort*. Insertion sort must in the worst case compare each element to every other element in the collection. It is therefore of quadratic time ($O(n^2)$) complexity. The process is frequently compared to the steps involved in sorting a hand of playing cards.

Setup This algorithm expects as input a linear collection of sortable keys.

Process For each element in the collection:

1. If the key has a left-side neighbor, compare it to that neighbor
2. If the key is smaller than its neighbor, swap them
3. Repeat from step 1 until either no more neighbor exists, or the key is larger than its neighbor

Sources

- *csharp/algorithms/searching/*
- *python_3/algorithms/*
- *bonus/fsharp/algorithms/*
- *bonus/rust/algorithms/searching/*

4.3.2 Shell sort

The *shell sort* algorithm utilizes insertion sort in a *divide-and-conquer* strategy. Shell sort is a $O(nlog^2n)$ time complexity algorithm.

Setup The algorithm expects as inputs a collection of sortable keys and a slice size. The collection will be cut up into slices according to the provided size. The results are stored into a buffer list or array.

Process

1. For each slice:
   (a) Run insertion sort on the slice
   (b) Once complete, concatenate the sorted slice to the buffer
2. Run insertion sort on the buffer

Sources

- *csharp/algorithms/shell_sort/*
- *python_3/algorithms/*

4.3.3 Bubble sort

The *bubble sort* algorithm lets elements in an unsorted collection 'bubble' up to the correctly sorted position in the resulting collection. Bubble sort is of quadratic time ($O(n^2)$) complexity.

Setup The algorithm expects as input an unsorted collection. It keeps track of an index indicating how many keys are already sorted.

Process Until the index is equal to the size of the collection:

1. Take a slice from the index to the end of the collection and increment the index
2. Take the rightmost key of the slice
3. If the key does not have a left-side neighbor, continue from step 1
4. If the left-side neighbor is smaller than the key, swap the keys
5. Continue from step 3

Sources

- *csharp/algorithms/bubble_sort/*
- *python_3/algorithms/*

4.3.4 Comb sort

The *comb sort* algorithm addresses a problem bubble sort faces when dealing with small values near the end of the unsorted input collection. Comb sort is a quadratic time ($O(n^2)$) complexity algorithm.

Setup The algorithm expects as input an unsorted collection of keys and a gap shrink factor. The first gap size is the length of the input collection multiplied by the shrink factor.

Process

1. Multiply the gap size by the shrink factor
2. If the gap size is smaller than or equal to 1 the algorithm is finished
3. Set an index to 0
4. If the index plus the gap size is larger than or equal to the length of the collection, continue from step 1
5. If the key at the current index in the collection is larger than the key at the index plus the gap size, swap those keys
6. Increment the index and continue from step 4

Sources

- *csharp/algorithms/comb_sort/*
- *python_3/algorithms/*

4.3.5 Counting sort

The *counting sort* algorithm involves generating a histogram of key frequencies. This algorithm is of $O(n+k)$ time complexity where k is the number of possible values in the min-max range.

Setup This algorithm expects as input an unsorted collection of keys. First, the maximum value in the collection is determined. Then, an array of the size of that value is populated with 0's.

Process

1. Using the keys in the collection as indices for the histogram array, count key frequencies
2. Set a sorting index to 0
3. For each index in the histogram, so long as the frequency is more than 0:
   (a) Set the value at the sorting index in the collection to the index in the histogram
   (b) Increment the sorting index
   (c) Decrease the frequency in the histogram

Sources

- *csharp/algorithms/counting_sort/*
- *python_3/algorithms/*

4.3.6 Bucket sort

The *bucket sort* algorithm divides the input collection into a series of 'buckets' depending on the minimum and maximum values in the input collection. It is of quadratic time $O(n^2)$ complexity. Bucket sort can use other algorithms to sort the buckets it generates.

Setup This algorithm expects as input an unsorted collection and a maximum size for the buckets. First, an array of buckets - each of which is its own array - is generated according to the minimum and maximum values in the collection, as well as the specified maximum bucket size. In addition, a list is created which will hold the result of the sorting process.

Process

1. For each key in the input collection:
   (a) Find the bucket index by subtracting the minimum value from the key
   (b) Append the key to its appointed bucket
2. For each bucket:
   (a) Run insertion sort or an alternative sorting algorithm on the bucket
   (b) Concatenate the sorted bucket to the sorted list

Sources

- *csharp/algorithms/bucket_sort/*
- *python_3/algorithms/*

4.3.7 Merge sort

This sorting algorithm splits a collection into many small collections. Each of these small collections is sorted and subsequently merged with another small collection. Since at each step the smaller collections that are being merged are already sorted, sorting the merged collection takes less time than sorting a badly sorted collection of the same size. *Merge sort* is of $O(nlogn)$ time complexity.

Setup This algorithm expects as input an an unsorted collection. In addition, it expects a left, middle and right pivot. Initially these will be set to the extremeties of the collection and its center.

Process

1. *Divide (split)* - Recursively split the collection by calling the divide function on both halves of the current input collection until the right index is no longer greater than the left index
2. *Conquer (merge)* - At the end of each resurfacing step of the division algorithm:
   (a) Take two segments, sort them by comparing the top elements of these collections and concatenating the smaller one to the sorted buffer
   (b) If either of the collections runs out, concatenate the remaining elements (which are already sorted) to the sorted buffer
   (c) When the merge is complete, continue surfacing from the division recursion

Sources

- *csharp/algorithms/merge_sort/*
- *python_3/algorithms/*
- *bonus/c/algorithms/*

4.3.8 Quick sort

Via the use of a pivot, the *quick sort* algorithm recursively partitions and conquers the input collection. Quick sort is of quadratic time ($O(n^2)$) complexity.

Setup This algorithm expects as input an unsorted collection and a start and end index. The start and end indices are initially set to the extremities of the collection.

Process As long as the end index is greater than the start index:

1. Set a left index to the start index and a right index to the end index
2. Set a valid pivot. In my implementation, the left index is also used as the pivot
3. As long as the left index is less than the right index:
   (a) Increment the left index until a key greater than the pivot is encountered
   (b) Increment the right index until a key less than the pivot is encountered
   (c) Swap the keys at the left and right indices in the collection
   (d) Increment the left index, decrement the right index

4. Recurse on the segment from the new start index to the new right index
5. Recurse on the segment from the left index to the new end index

Sources

- *csharp/algorithms/quick_sort/*
- *python_3/algorithms/*

4.3.9 Monkey sort

Imagine having an unsorted deck of cards, then throwing it in the air and picking it up to see if the new distribution is sorted. If it's not, then you repeat the process. This approach to sorting is a description of *monkey sort*. It's a highly inefficient sorting algorithm, in which a random distribution of the collection is made every time. Subsequently, the new distribution goes through a process that verifies if it is sorted. If it isn't, a new distribution is made. Since this algorithm may never yield a sorted distribution, the time complexity is unbounded ($O(\infty)$). The primary point of this algorithm is to demonstrate the importance of proving that your algorithm can reasonably be assumed to complete within an acceptable time-frame.

Sources

- *python_3/algorithms/*

4.4 Shortest path determination

In this section, each of the following algorithms is assumed to operate on the following graph:

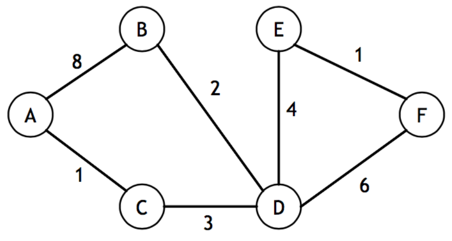

Figure 11: A weighted graph

When recording this graph as data, letters are replaced with 0-based indices, such that $A = 0, B = 1, C = 2, \ldots$.

4.4.1 Dijkstra

Dijkstra's algorithm finds all shortest path distances between a *source* vertex and all other vertices in a graph. Dijkstra's algorithm runs in quadratic time ($O(n^2)$). It's possible to optimize Dijkstra's algorithm using a *Fibonacci heap* as a priority queue, although my implementation does not include one.

Setup The algorithm accepts as inputs a graph and a source vertex. First it sets up an array which will hold the result distances. This array is initially populated with the value ∞, with the exception of the source vertex's distance to itself; which is 0.

Process All unvisited vertices are stored in a list, so that the algorithm may prevent itself from repeatedly processing previously visited vertices. As long as unvisited vertices remain, the algorithm will select the closest unvisited vertex and remove it from the list of unvisited vertices. For each neighboring vertex, the algorithm will check if the newly found distance to it is smaller than the currently recorded vertex. If it is, the new distance will be recorded in the distances array. When no unvisited vertices remain, the distances array will be populated with all shortest path distances.

Chain With a slight modification it becomes possible to also track the actual paths, rather than just the distances. During the setup phase, the algorithm should also create an array which will hold the chain information. Initially, all values will be set to null. Whenever a shorter distance is found, the current vertex will be recorded in the chain using the neighbor's ID as the index.

Shortest path The *shortest path* operation can compute the actual shortest path when given two vertex ID's and a precomputed chain. It is a linear time ($O(n)$) algorithm. The result of the shortest path operation will be a list containing each step of the path. The operation starts at the target vertex. As long as the currently inspected vertex is not null, it is appended to the existing path. The next vertex to inspect may be located by using the current vertex ID as the index of the next path vertex in the chain.

Sources Floyd-Warshall implementations are provided at the following locations:

- *csharp/algorithms/dijkstra/*
- *python_3/algorithms/*

4.4.2 Floyd-Warshall

Rather than Dijkstra's focus on a single source, *Floyd-Warshall* finds the shortest path between each vertex and all other vertices in a graph. Unlike Dijkstra's algorithm, Floyd-Warshall supports negative weights. Initially one might suspect that this algorithm somehow runs Dijkstra on each vertex, which would be rather inefficient. Fortunately, Floyd-Warshall uses a more clever approach.

Setup The algorithm expects a graph, which I've provided in the form of a weighted adjacency list. An initial 'guess' is prepared by setting up a matrix of distances between vertices in the graph. This matrix will at first contain the value ∞ for each distance. To improve the initial guess, the matrix is populated with the known distances between adjacent vertices.

Process There are three nested *for* loops, using the variables i, j, and k. At each iteration, i and j are two vertices between which the algorithm is trying to find the shortest path. The most important aspect of the algorithm actually relies on k. At each iteration, k represents a vertex between i and j that potentially yields a shorter path from i to j. This means that each time we find a shorter path, we'll have to update the currently recorded value in our distance matrix. Since we have three nested *for* loops that each scale according to the amount of vertices in the graph, it naturally follows that our algorithm is of cubic time ($O(n^3)$) complexity.

Sources Floyd-Warshall implementations are provided at the following locations:

- *csharp/algorithms/floyd_warshall/*
- *python_3/algorithms/*