2018

Algorithms Notes

UNIVERSITY OF CAMBRIDGE, PART IA

ASHWIN AHUJA

Table of Contents

Algorithms

Introduction

An algorithm is a systematic recipe for solving a problem. Therefore, we have to precisely specify the problem and before we can consider it complete, we need a proof that it works (correctness) and an analysis of the performance.

By having complete solutions to common sub-problems, it allows us to simplify more complicated problems which we have split into the requisite sub-problems to be combined together.

For an algorithm, we need to consider:

1. Strategy
2. Actual algorithm
3. Data structure(s)
4. Proof of correctness of the algorithm
5. Proof of time complexity analysis algorithm
6. Other important ranking consideration between algorithms completing the same problem
    a. For sorting this may include in-place-ness and stability.

Sorting

Documentation, Preconditions and Postconditions

Precondition: Request, specifying what the routine expects to receive from its caller
Postconditions: Promise, specifying what the routine will do for its caller (provided the precondition is met)

Together, the precondition and postconditions form some kind of "contract" (this is terminology of Bertrand Meyer) between the routine and its caller.

Correctness

When we have considered an algorithm, we must always consider if the algorithm is correct. In order to do this, we can do a few things:

1. Specify the objects as clearly as possible
    a. **This allows us to actually say if the algorithm is correct or not – without a specification, we do not know whether something is correct or not.**
2. Reduce a large problem to a sequence of smaller sub problems where we can apply mathematical induction.
3. Assertions
    a. These act as stepping stones which mean we can assume that we have reached a certain point in the algorithm with everything being as expected.
    b. We can prove each assertion for each point with the final assertion being that the entire problem has been solved.
    c. It is particularly useful to have an assertion at the start of each significant loop, saying how the previous loop affected the state. Therefore, we can prove this inductively (with a base case going into the first loop and then showing the inductive step over one loop).

Computational Complexity

We make a number of assumptions which help us to simplify the process to perform the cost estimation. The most commonly used assumptions are:

1. Only worry about the worst possible amount of time that the activity could take
2. Rather than measuring absolute computing times, we only look at rates of growth and we ignore constants.
3. Any finite number of exceptions to a cost estimate are unimportant as long as the estimate is valid for all large enough values of n.
4. We do not restrict ourselves to just reasonably values of n or apply any other reality checks. The cost estimation will be carried through as an abstract mathematical activity.

Worst, average and amortized costs

Usually the simplest way of analysing an algorithm is to find the worst case performance. This is very important where we must often consider when the algorithm is at its worst and may take the most time that it would ever take. This is particularly important for real-time applications, where we want to consider what would happen in the worst case.

Average case analysis should generally be more interesting to most people. However, before useful average cost analysis can be performed, you need a model for the probabilities of all possible inputs. If in some particular application, the distribution of inputs is significantly skewed, then analysis based on uniform probabilities might not be valid. For worst case analysis, it is only necessary to study one limiting case – therefore being mathematically much harder.

Amortized analysis is applicable where the data structure supports a number of operations and these will be performed in sequence. Quite often the cost of any particular operation will depend on the history of what has been done before and sometimes a plausible overall design makes most operations cheap at the cost of occasional expensive internal reorganisation of the data. In amortized analysis, we treat the cost of this reorganisation as the joint responsibility of all the operations previously performed and provides a firm basis for seeing if it was worthwhile.

Big-O, Θ and Ω notation

A function f(n) is said to be O(g(n)) if there exists constants k > 0 and N > 0, such that:

$$0 \leq f(n) \leq kg(n) \; for \; n > N$$

Therefore, effectively g(n) provides an upper bound that, for sufficiently large values of n, f(n) will never exceed, except for what can be compensated by a constant factor.

A function f(n) is said to be Θ(g(n)) if there are constants $k_1 > 0$, $k_2 > 0$, N > 0, such that:

$$0 \leq k_1 g(n) \leq f(n) \leq k_2 g(n) \; for \; n > N$$

In other words, for sufficiently large values of n, the functions f() and g() agree within a constant factor. This constraint is much tighter than the Bug-O.

We use Ω as the dual of O() to provide a lower bound. Therefore: $f(n) \epsilon \Omega(g(n))$ means that f(n) grows at least like g(n)

It is important to state that providing a bound for a function does not mean that it is a good bound for it. Some people, therefore use O() and Ω() to describe bounds that might be tight, but o() and ω() for bounds that are definitely not tight.

	If...	then $f(n)$ grows ... $g(n)$.	$f(n)$ ... $g(n)$
small-o	$f(n) \in o(g(n))$	strictly more slowly than	$<$
big-o	$f(n) \in O(g(n))$	at most as quickly as	$\leq$
big-theta	$f(n) \in \Theta(g(n))$	exactly like	$=$
big-omega	$f(n) \in \Omega(g(n))$	at least as quickly	$\geq$
small-omega	$f(n) \in \omega(g(n))$	strictly more quickly than	$>$

N.B. Very common to say f(n) = O(g(n)) instead of f(n) ϵ O(g(n))

Logarithms: For the purposes of time complexity, we consider logs as a singular concept, i.e. $O(\log_2(m)) = O(\log_5(m)) = O(\lg(m))$. This is clearly because they are a scalar away from one another.

Models of Memory

We always assume that computers to run algorithms will always have enough memory and that the memory can be arranged in a single address space so that one can have unambiguous memory addresses and pointers.

Obviously, this is not true, and in fact, we normally also assume that any element of an array can be accessed in unit time, however large the array gets – associated assumption is that integer arithmetic operations needed to computer array subscripts can also be done at unit cost.

Today, what with caches, obviously, not all memory retrieval is equivalent, and some may be instant, and some may take tens or hundreds of clock cycles.

Models of Arithmetic

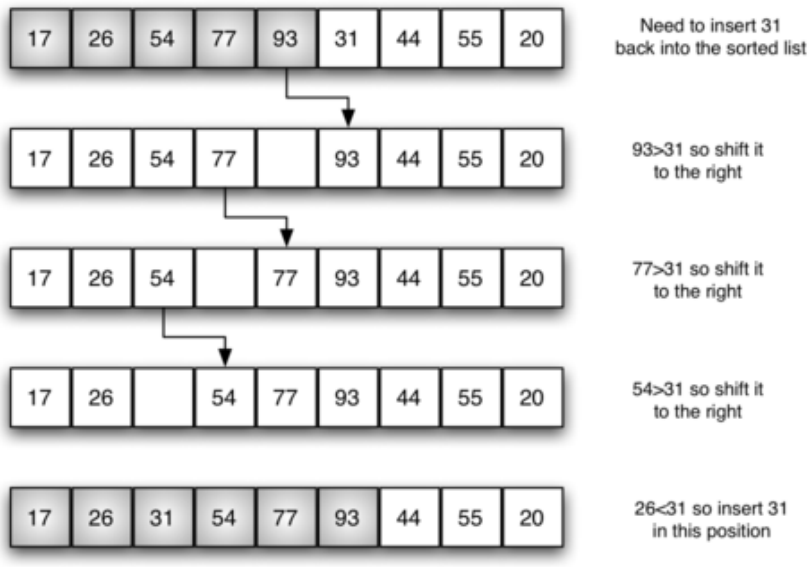
Normal model for computer arithmetic is that each arithmetic operation takes unit time, irrespective of the values of the numbers being combined and regardless of whether fixed or floating-point numbers are involved. In fact, as long as are using Θ notation, we swallow up constant factors in timing.

We must sometimes consider a theoretical problem where numbers may be many orders larger than native computer arithmetic will support directly. Here, we need to show the cost analysis to explicitly consider how much extra work which will be involved in doing the multiple precision arithmetic, and then timing estimates will generally depend, not only on the number of values involved in a problem but the number of bits needed to specify each value.

Insertion Sort

```
0  def insertSort(a):
1      """BEHAVIOUR: Run the insertsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for i from 1 included to len(a) excluded:
10         # ASSERT: the first i positions are already sorted.
11
12         # Insert a[i] where it belongs within a[0:i].
13         j = i - 1
14         while j >= 0 and a[j] > a[j + 1]:
15             swap(a[j], a[j + 1])
16             j = j - 1
```

Recursively move an item from the unsorted portion of the array and move it to the sorted portion.




```
public static int[] insertionSort(int[] unSortedList)
{
    for (int i = 1; i < unSortedList.length; i++)
    {
        int j = i-1;
        while(j >= 0 && unSortedList[j] > unSortedList[j+1])
        {
            int temp = unSortedList[j];
            unSortedList[j] = unSortedList[j+1];
            unSortedList[j+1] = temp;
            j--;
        }
    }
    return unSortedList;
}
```

The outer loop of line 9 (pseudocode) is executed exactly n-1 times (regardless of the values of the elements in the input array), while the inner loop is executed a number of times that depends on the number of swaps to be performed.

In the worst case, during the ith invocation of the outer loop, the inner loop will be performed i times. Therefore for the whole algorithm, the execution number won't exceeed the nth triangular number (1 + 2 + … + n) which is equal to ½ n(n+1) = $O(n^2)$

Optimal Sorting

If I have n items in an array and I need to rearrange them in ascending order, whatever the algorithm there are two elementary operations that I can plausibly expect to use repeatedly in the process. The first takes two items and compares them to see which should come first. The second swaps the contents of two array locations.

Assertion 1 (lower bound on exchanges): If there are n items in an array then Θ(n) exchanges always suffice to put the items in order. In the worst case, Θ(n) exchanges are actually needed.

Proof: Identify the smallest item present, if it not already in the right place then one exchange moves it to the start of the array. A second exchange moves the second smallest, etc. Therefore, after n-1, we will be done therefore equal to Θ(n) we would be guaranteed to be done.

Assertion 2 (lower bound on comparisons): Sorting by pairwise comparisons, assuming that all possible arrangements might actually occur as an input, necessarily costs at least Ω(n lg n)

Proof: There are n! permutations of n items and in sorting, we identify one of these. In each test, we half the possible number of correct arrangements. Therefore, we need $\log_2(n!)$ tests.

Stirling's formula tells us that n! is approximately equal to n^n therefore log (n!) is approximately equal to n lg n

More importantly, $lg(n!) \geq nlgn$ (for O notation) simply by:

$$\lg(n!) = \lg(n) + \lg(n-1) + \cdots + \lg(1) \leq \mathrm{n} \lg n$$

Therefore, lg(n!) is bounded by n lg n. Conversely, since the lg function is monotonic, the first n/2 terms from lg n to lg n/2 are all greater than or equal to lg (n/2) = lg n – lg 2 = lg n – 1. Therefore:

$$\lg(n!) \quad \geq \quad \frac{n}{2}(\lg n - 1) + \lg(n/2) + \ldots + \lg(1) \quad \geq \quad \frac{n}{2}(\lg n - 1),$$

Thus, when n is large enough n lg n is bounded by k lg n!. Therefore lg(n!) = Θ(n lg n)

Selection Sort

The idea of the proof showing Θ(n) swaps acts as the strategy for this sorting strategy. In this case, for each iteration, we need to find the smallest item of the array. Here we can scan

linearly through the sub-array, comparing each successive item with the smallest one so far. If there are m items to scan, then finding the minimum clearly costs m-1 comparisons. The whole selection-sort does this on a sequence of sub-arrays of n, n-1, … , 1. Therefore, total number of comparisons and exchanges = $\Theta(n^2)$ by the summation of a simple arithmetic progression (triangular number).

```
0  def selectSort(a):
1      """BEHAVIOUR: Run the selectsort algorithm on the integer
2      array a, sorting it in place.
3
4      PRECONDITION: array a contains len(a) integer values.
5
6      POSTCONDITION: array a contains the same integer values as before,
7      but now they are sorted in ascending order."""
8
9      for k from 0 included to len(a) excluded:
10         # ASSERT: the array positions before a[k] are already sorted
11
12         # Find the smallest element in a[k:END] and swap it into a[k]
13         iMin = k
14         for j from iMin + 1 included to len(a) excluded:
15             if a[j] < a[iMin]:
16                 iMin = j
17         swap(a[k], a[iMin])
```

Binary Insertion Sort

As with insertion sort, we imagine some sort of partition and place the next item into the sorted part of the array. In order to find where in the initial part of the array, we can do a binary search, which will take $\lceil \lg k \rceil$ where k is the size of the sorted array. Then we can drop the item in place using at most k exchanges.

Therefore, number of comparisons =

$$\lceil \lg 1 \rceil + \lceil \lg 2 \rceil + \cdots + \lceil \lg n - 1 \rceil$$

This is bounded by:

$$\lg 1 + 1 + \lg 2 + 1 + \cdots + \lg n - 1 + 1$$

Thus, this is bounded by:

$$\lg\big((n-1)!\big) + n = O(\lg(n)\,!) = O(n\ lg\ n)$$

This means that we effectively attain the lower bound for general sorting but we still have very high (quadratic) data movement costs. Even if the swap were a constant amount cheaper than a comparison, therefore, the overall asymptotic cost is $O(n^2)$

```
def binaryInsertSort(a):
    """BEHAVIOUR: Run the binary insertion sort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    for k from 1 included to len(a) excluded:
        # ASSERT: the array positions before a[k] are already sorted

        # Use binary partitioning of a[0:k] to figure out where to insert
        # element a[k] within the sorted region;

        ### details left to the reader ###

        # ASSERT: the place of a[k] is i, i.e. between a[i-1] and a[i]

        # Put a[k] in position i. Unless it was already there, this
        # means right-shifting by one every other item in a[i:k].
        if i != k:
            tmp = a[k]
            for j from k - 1 included down to i - 1 excluded:
                a[j + 1] = a[j]
            a[i] = tmp
```

Bubble Sort

Similar to insertion sort and it is very easy to implement. It consists of repeated passes through the array during which adjacent elements are compared and, if out of order, swapped. The algorithm terminates as soon as a full pass requires no swaps.

```
def bubbleSort(a):
    """BEHAVIOUR: Run the bubble sort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    repeat:
        # Go through all the elements once, swapping any that are out of order
        didSomeSwapsInThisPass = False
        for k from 0 included to len(a) - 1 excluded:
            if a[k] > a[k + 1]:
                swap(a[k], a[k + 1])
                didSomeSwapsInThisPass = True
    until didSomeSwapsInThisPass == False
```

Like insertion sort, this algorithm has quadratic costs in the worst case, but it terminates in linear time on inputs that was already sorted. This is clearly an advantage over Selection sort.

Mergesort

Given a pair of sorted sub-arrays, each of length n/2, merging their elements into a single sorted array is easy to do in O(n). This clearly just keeps taking the lowest element from the sub-array that has it.

Therefore, we can sort in this sense, split the input array into two halves and sort them recursively, stopping when the chunks are so small that they are already sorted, and then merge the two sorted halves into one sorted array.

```
0  def mergeSort(a):
1      """*** DISCLAIMER: this is purposefully NOT a model of good code
2     (indeed it may hide subtle bugs---can you see them?) but it is
3     a useful starting point for our discussion. ***
4
5      BEHAVIOUR: Run the merge sort algorithm on the integer array a,
6      returning a sorted version of the array as the result. (Note that
7      the array is NOT sorted in place.)
8
9      PRECONDITION: array a contains len(a) integer values.
10
11     POSTCONDITION: a new array is returned that contains the same
12     integer values originally in a, but sorted in ascending order."""
13
14     if len(a) < 2:
15         # ASSERT: a is already sorted, so return it as is
16         return a
17
18     # Split array a into two smaller arrays a1 and a2
19     # and sort these recursively
20     h = int(len(a) / 2)
21     a1 = mergeSort(a[0:h])
22     a2 = mergeSort(a[h:END])
23
24     # Form a new array a3 by merging a1 and a2
25     a3 = new empty array of size len(a)
26     i1 = 0 # index into a1
27     i2 = 0 # index into a2
28     i3 = 0 # index into a3
29     while i1 < len(a1) or i2 < len(a2):
30         # ASSERT: i3 < len(a3)
31         a3[i3] = smallest(a1, i1, a2, i2) # updates i1 or i2 too
32         i3 = i3 + 1
33     # ASSERT: i3 == len(a3)
34     return a3
```

Compared to other sorting algorithms so far, this one hides several subtleties, many to do with memory management issues:

1. Merging two sorted sub-arrays is most naturally done by leaving the two input arrays alone and forming the result into a temporary buffer as large as the combination of the two inputs. This means that, unlike the other algorithms seen so far, we cannot sort an array in place, we need additional space.
2. The recursive calls of the procedure on the sub-arrays may involve additional acrobatics in languages where the size of the arrays handled by a procedure must be known in advance.
3. Merging the two sub-arrays is conceptually easy but coding it naively may fail on boundary cases.

In order to evaluate the running cost, we can write a recurrence relation:

$$f(n) = 2f\left(\frac{n}{2}\right) + kn; f(1) = 1$$
$$= 2^2 f\left(\frac{n}{2^2}\right) + 2kn$$
$$= 2^z f\left(\frac{n}{2^z}\right) + zkn$$

We hit f(1) when n = 2^z

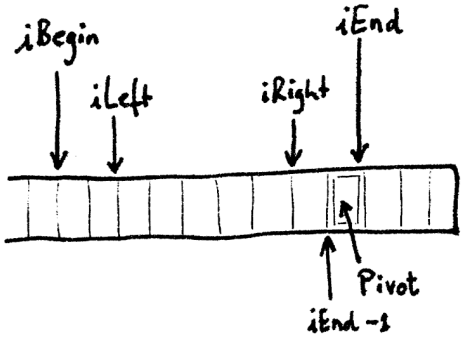
$$= nf(1) + \text{kn} \lg n$$
$$\cong O(n \lg n)$$

Therefore, we confirm that Mergesort guarantees the optimal cost, is relatively simple and has low time overheads. However, it requires extra space to hold the partially merged results. The implementation is trivial if one has an extra n space and still easy-ish with n/2.

An alternative is to run the algorithm bottom-up, doing away with the recursion. Then you group the sorted pairs two by two and sort (by merging) each pair. Then group the sorter pairs, merging them, etc. Unfortunately, although this eliminates the recursion, this still requires O(n) additional storage, because to merge two groups of k elements into a 2k sorted group, you need an auxiliary area of k cells – *Move the first group into the extra space, then put in place in first group then the second group (guaranteed to have space for it).*

Quicksort

The core idea of Quicksort is to select some value from the input and use that as a pivot to split the other value into two unsorted subsets: those smaller and those larger than the pivot. Then, quicksort can be recursively applied to these subsets.



In order to do this, we can work in-place. To partition, we scan the array from both ends to partition it into three regions. We arbitrarily pick the last element in the range to be the pivot. Then, iLeft starts at iBegin and moves right, while iRight starts at iEnd-1 and moves left.

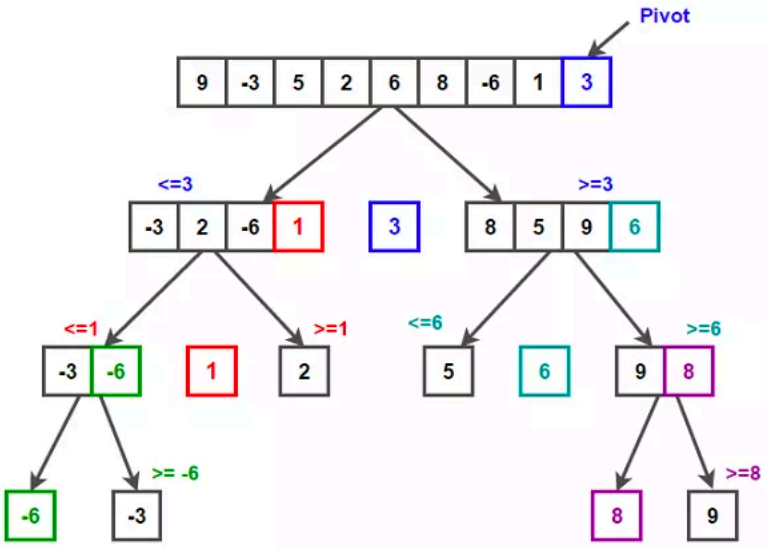
We have the following invariants:

1. iLeft ≤ iRight
2. array[iBegin : iLeft] only has elements ≤ pivot
3. array[iRight : iEnd – 1] only has elements > pivot

So long as iLeft and iRight have not met, we move iLeft as far right as possible and iRight as far left as possible without violating their invariance. Once they stop, if they haven't met, it means that array[iLeft] > pivot and array[iRight] ≤ pivot. Therefore, we swap these two elements. Then we repeat the same process, pushing iLeft and iRight as far towards each other as possible, swapping array elements when the indices stop and continuing until they touch.

When they touch, we put the pivot in the right place, by swapping array[iRight] and array[iEnd – 1]

We can then recursively run Quicksort on the two smaller sub-arrays which are left.



Performance

In the ideal case we partition into two equal parts. Then the total cost of Quicksort satisfies the recurrence f(n) = 2f(n/2) + kn and f(1) = 1. Therefore it grows as O(n lg n).

In the worst case, when we learn that the item is the lowest or highest when we partition (it is already sorted or not sorted), then we get f(n) = f(n-1) + kn, therefore f(n) = $O(n^2)$

In order to get an average cost, we consider every possible pivot choice. This is found by:

$$f(n) = kn + \frac{1}{n}\sum_{i=1}^{n}\big(f(i-1) + f(n-i)\big)$$

$$\{where\ kn\ is\ the\ cost\ of\ partitioning\}$$
$$= \Theta(n \lg n)$$

This shows how quicksort has a good average performance, but has an uncommon, but unsatisfactory worst-case performance. Therefore, it should not be used in applications where the worst-case costs could have safety implications.

Variants of Quicksort

1. Using median (of-X) as partition point
    a. This means that you are unlikely to get to the worst case of the $O(n^2)$
2. Insertion sort below a threshold
    a. Often it is better to insertion sort when the number of numbers left is very reduced.
    b. This means we can avoid some unnecessary recursion

Median and order statistics using Quicksort

Sometimes, we don't need a sorted array, as much as a partially sorted array, eg to get the nth element. Therefore, we could sort the entire array and read off what we want – this would be O(n lg n), however, this is clearly wasted effort.

We can instead only recurse on the half of the array which is relevant, after the array is split. Therefore, in the best case:
F(n) = F(n/2) + kn => O(n)

However, in the worst case, it is still quadratic:
F(n) = F(n-1) + kn => $O(n^2)$

There is a better method which can make this happen in O(n) for all arrays, but this algorithm does not need to be considered in the course.

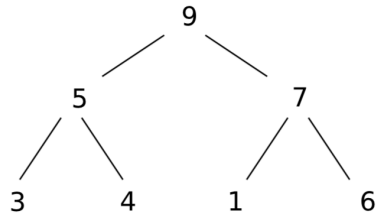
Heapsort

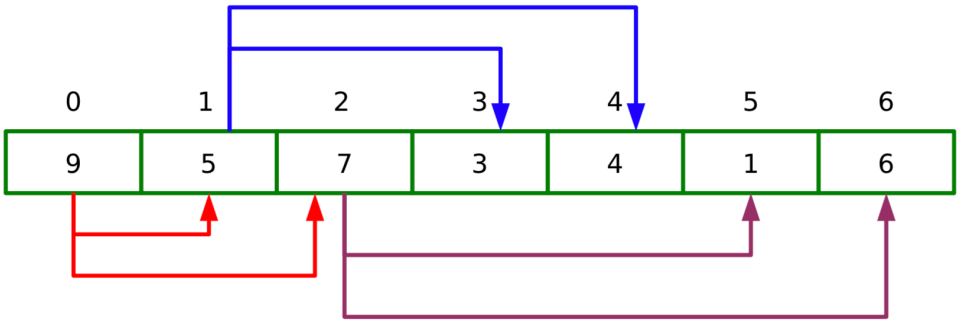
Heap sort allows us to sort in place and guarantees O(n lg n) for any input – although the constant of proportionality is larger than for quicksort

Max-Heap
It is a simple binary tree with 2 rules:

1. The value of any child must be less than the value of its parent.
2. The tree must be almost full

We use an array to represent the tree with the following way:

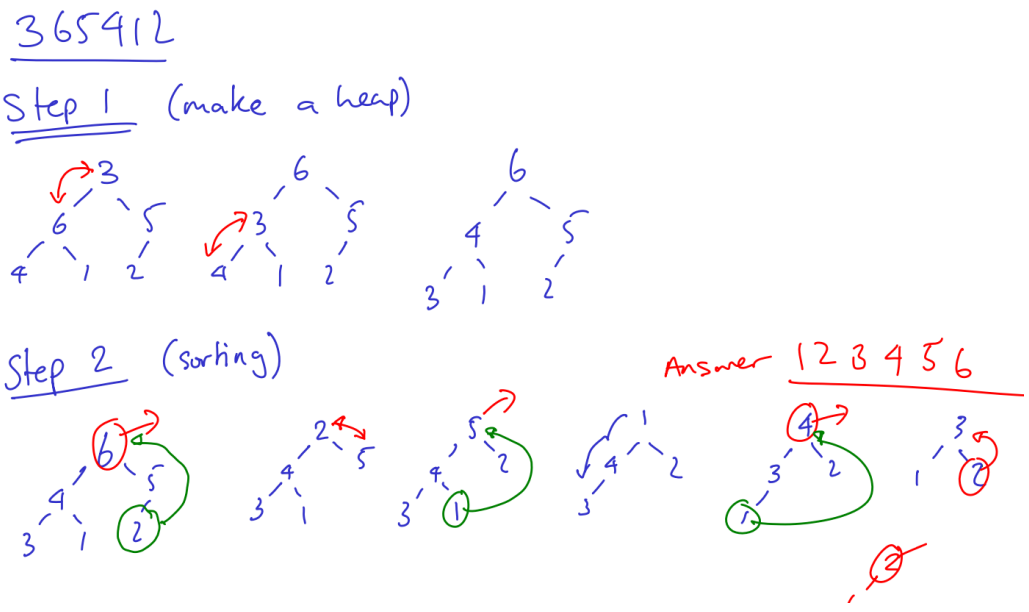




The children of the node at x can be found at 2x + 1 and 2x + 2. The array representation forces the second rule since we can't have array gaps except right at the end.

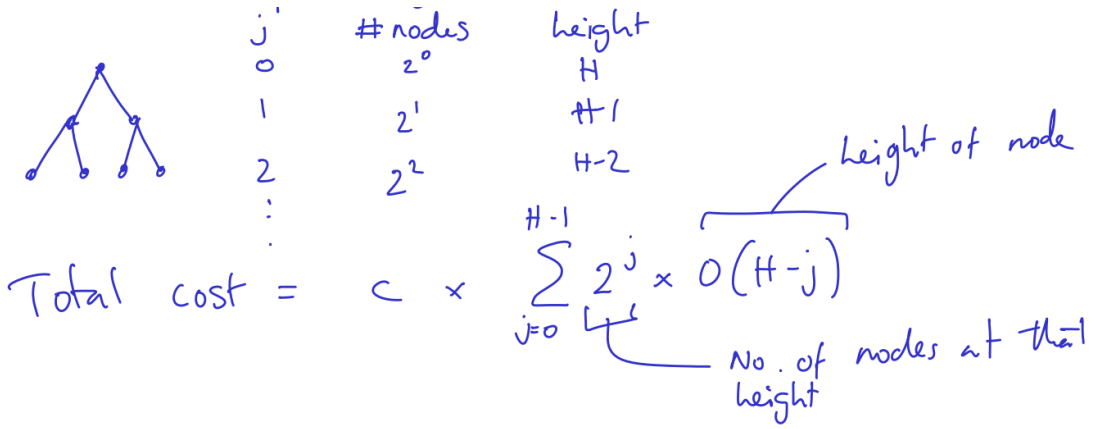
Heapsort largely follows this path:

1. Make your data into a heap
2. For (k = n-1 … 0)
    a. Swap top element and kth element
    b. Heapify – make array 0 … k-1 a valid heap



Complexity of making a heap

In order to create a heap, we build from the bottom up. When we get to a node, we know its children are valid heaps. In the worst case, for any node, we have to filter it all the way down (and replace with highest child). This is O(h) where h is the height of the current node.



$$c\sum_{j=0}^{H-1} 2^j O(H-j) = d\sum_{j=0}^{H-1} 2^j (H-j) = d\,2^H \sum_{j=0}^{H-1} \left(\frac{1}{2}\right)^{H-j} (H-j)$$

$$= d\frac{2^H \frac{1}{2}}{\frac{1}{4}} (using\ Standard\ Result) = O(2^H) = O\left(2^{\lceil \lg n \rceil}\right) = O(n)^1$$

Therefore, it takes linear time to create the heap.

Complexity of fixing heap

In the worst case, we have to filter the root back down all the way – O(h) where h changes with the extractions:

$$Cost = \lceil \lg n \rceil + \lceil \lg n - 1 \rceil + \lceil \lg n - 1 \rceil + \cdots + \lceil \lg 1 \rceil = O(n \lg n)$$

[1] Important assumption is this is as H goes to infinity since we use the standard result: $\sum_0^\infty ax^a = \frac{x}{(1-x)^2}$

```
def heapSort(a):
    """BEHAVIOUR: Run the heapsort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    # First, turn the whole array into a heap
    for k from floor(END/2) excluded down to 0 included: # nodes with children
        heapify(a, END, k)

    # Second, repeatedly extract the max, building the sorted array R-to-L
    for k from END included down to 1 excluded:
        # ASSERT: a[0:k] is a max-heap
        # ASSERT: a[k:END] is sorted in ascending order
        # ASSERT: every value in a[0:k] is <= than every value in a[k:END]
        swap(a[0], a[k - 1])
        heapify(a, k - 1, 0)


def heapify(a, iEnd, iRoot):
    """BEHAVIOUR: Within array a[0:iEnd], consider the subtree rooted
    at a[iRoot] and make it into a max-heap if it isn't one already.

    PRECONDITIONS: 0 <= iRoot < iEnd <= END. The children of
    a[iRoot], if any, are already roots of max-heaps.

    POSTCONDITION: a[iRoot] is root of a max-heap."""

    if a[iRoot] satisfies the max-heap property:
        return
    else:
        let j point to the largest among the existing children of a[iRoot]
        swap(a[iRoot], a[j])
        heapify(a, iEnd, j)
```

Stability

When we have which consists of a key (about which the ordering is based) and the data which is attached to the data, it is important to consider what should happen when the key is equal. In stable ordering, the order of items in the input must be preserved in the output where the keys are equal, whereas in non-stable orderings this ordering may not be maintained.

If stability is required but not delivered by the sorting algorithm, we can add another field to each of the record with the original position. While sorting, when breaking ties, check this field to see where the items should be. Then, any arbitrary sorting method will rearrange the data in a stable way, though this clearly increases space and time overheads a little.

Counting Sort

Every sorting algorithm we have considered so far has been a comparison sort, we use the less-than or greater-than operator to compare multiple elements and decide their orderings.

However, it is possible to sort without the comparison operators, using **distribution sorts**

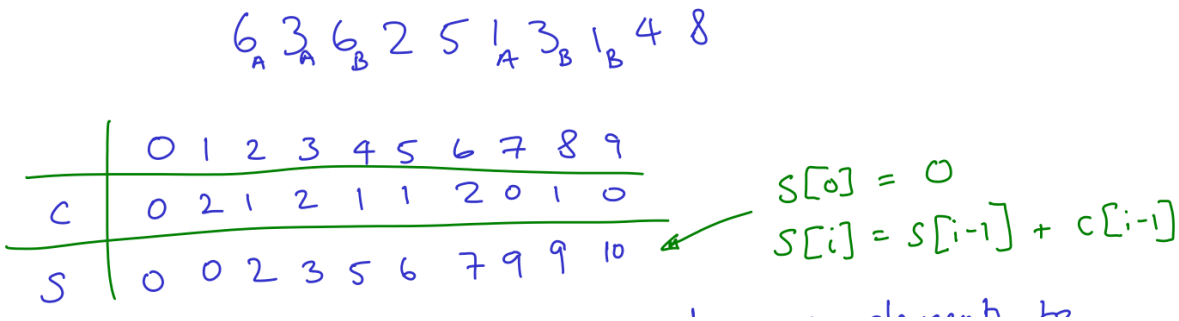
In counting sort, we can sort when we have a specific range of values that could be in the input.

1. **Find the min and the max of the input**
2. **Build the count array**
    a. **The number itself and the number of iterations of the number**
3. **Walk through the count array, reading out the numbers in order**

This clearly has cost of O(n + k) and space of O(k) where k = max – min

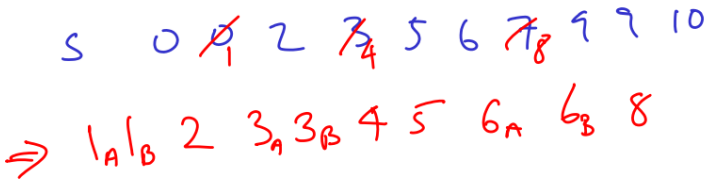
Stable Counting Sort

It is also able to develop a counting sort where each value has some satellite data. In order to do this, we build a count array (as before) **and start array**, describing where each value starts in the output.



We create the start array as described by the formula on the right.

Now, we walk through the original input and move the elements to the position described by s[value]. **Then we increment s[value]**



In addition to being stable, it still has a cost of O(n+k) and space of O(n+k) – since we need to store the original input when we are walking through it.

Alternatively, to do the stability, we can do it as we describe for a general method and take a pass through the sorted (using the unstable) and where the keys match, lookup in the original list which appeared first.

Bucket Sort

Assuming we have n keys uniformly distributed over some known range, in bucket sort we create an array of n linked lists. A linear scan adds each key to the list at index which fits the bucket. Say the keys were distributed between 0 and 1, then it would be at index kn for key k.

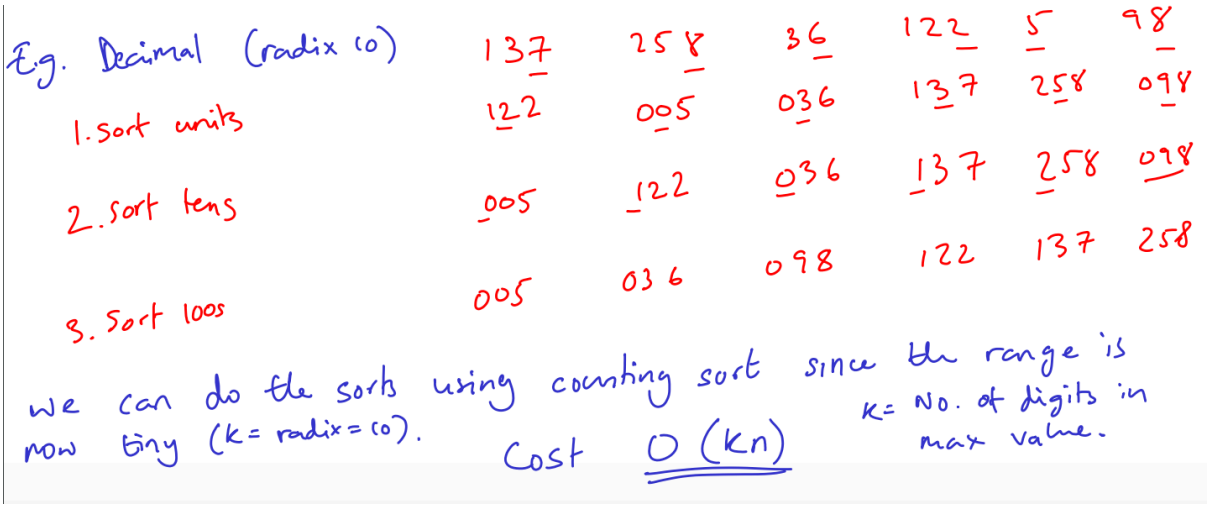
We expect the length of each list to be one, but there will some of length 0, some of length of 1, etc. If a list contains more than 1 it is then sorted – we can do this using any algorithm since the list should be very short (insertion sort for example).

Even though the $O(n^2)$ algorithm is used for the sorting, this is our worst case, however, we find that the average case is O(n)

It clearly only takes space O(n)

Radix Sort

When the range of the numbers is much larger than the number of things to sort, counting sort is very bad. Instead we sort by digits, using another sorter, which must be stable. It works best using the lowest digit first.



Strategies for Algorithm Design

Divide and Conquer

1. **DIVIDE:** split the problem into parts – almost always two
    a. If the instance is very small, you should instead solve it.
    b. The subproblems should be of a similar size
2. **CONQUER:** Use recursion to solve the smaller problems.
3. **COMBINE:** Create a solution to the final problem by using information from the solution of the smaller problems.

Dynamic Programming

Dynamic Programming is Divide and Conquer where the subproblems overlap. This means that we save lots of effort and saves us doing the same problem a large number of times.

In bottom-up, we develop a solution from the start circumstances that we have and work forwards, often using an array to list the number of ways we can do something to reach a certain point. This method of using an array to do this is called **tabulation**

In top-down, we can solve it using recursion. For example for a problem where we wanted to find f(9) where f(x) = f(x-1) + f(x-2) (Fibonacci numbers) we could say f(9) = f(8) + f(7). We could recursively find f(8) continuing until we hit a base case (f(0) = 0, f(1) = 1) and store this information in a table so when we want to find f(7) we already have this saved. When we save results only when they are directly called for (rather than starting at the bottom and building up), we call it **memoisation.** The savings can be very significant for doing this!

Optimisations

Dynamic Programming comes into its own when the problem requires some sort of optimisation – perhaps requires the best solution.

For a problem hence:

Consider an elevator with 3 buttons: button *1* goes up one floor; *2* goes up two floors; *3* goes up three floors. How many ways are there to move up F floors without going down at any point[2]?

Where instead of considering the number of ways, we want to see the solution that takes us up using the fewest button presses. We look for a recursive solution that allows us to reuse solutions:

We just need to add to our algebra to define N_i as the minimum number of button presses to move up i floors. Then

$$N(i) = 1 + \min \begin{cases} N_{i-1} \\ N_{i-2} \\ N_{i-3} \end{cases}$$

Finally, in order to get the path, you can return this using the recursion, when using a top-down memoised method.

When Dynamic Programming should be used

1. When there exist many choices, each with its own score to be minimised or maximised
2. The number of choices is exponential in the size of the problem (not brute-forceable)
3. Structure of the optimal solution is such that it is composed of optimal solutions to smaller problems.
4. There is overlap – the subproblems occur many times.
5. You can write a recurrence relation to describe the optimal solution to a sub-problem in terms of optimal solutions to smaller sub-problems.

Greedy Algorithm

- Always perform whichever operation contributes the most in a single step (**the locally optimal choice**)
- This is simple to implement and understand
    - Has this general pattern
        - **1: Cast the problem as one where we make a greedy choice and are left with one smaller problem to solve**
        - **2: Prove that the greedy choice is always part of an optimal solution**

        - **3: Prove that there's optimal substructure, i.e. that the greedy choice plus an optimal solution of the subproblem yields an optimal solution for the overall problem.**
- But it doesn't always optimize fully
    - We can perform greedy algorithm when it meets two requirements
        - **1: Greedy choice property:** You can make a locally optimal choice based only on past choices. This means you need a way of quantifying the value of each choice
        - **2: Optimal Substructure:** The optimal solution(s) contain optimal solutions to subproblems.

Other

1. **Recognise a variant on a known problem**
2. **Reduce to a simpler problem**
3. **Backtracking**
    a. If the algorithm requires a search, backtracking can be used.
    b. This splits the design of the procedure into two parts. The first part just ploughs ahead and investigates what it thinks is the most sensible path to explore, while the second backtracks when needed.
    c. The first path will reach a dead end and when it does, it backtracks.
4. **MM Method**
    a. Throw the problem to a large number of people (or to a random array of random changes) and wait a period of time and see which solution works.
5. **Seek wasted work in a simple method**
    a. Firstly design a simple algorithm to solve a problem then analyse it to the extent that the critically costly parts of it can be identified.
    b. Then we can seek to eliminate these elements and improve the algorithm.
6. **Seek formal mathematical lower bound**
    a. By finding a properly proved lower bound, we can prevent wasted time seeking improvement where none is possible.
7. **Brute Force**
    a. Generate all possible solutions and test each of them to see if it is the correct solution until you find it.
    b. You should only use when the number of possibilities is very low.

Data Structures

An Abstract Data Type is a mathematical model of a data type – a logical description that does not concern itself with the specific implementation.

A data structure is a concrete implementation of a data type. An ADT is a user of the type and a data structure is for the implementer of the type.

This whole idea employs the notion of data hiding and encapsulation, where how the ADT is actually working is completely irrelevant and does not need to be considered.

Primitive Data Types

We can generally assume that we have a set of primitive values that exist

1. Boolean
2. Character
3. Integer

4. Real
    a. We avoid considering overflows in reals and integers
5. Arrays
    a. **An ordered collections of predefined number of elements of the same array time.**
6. Pointers

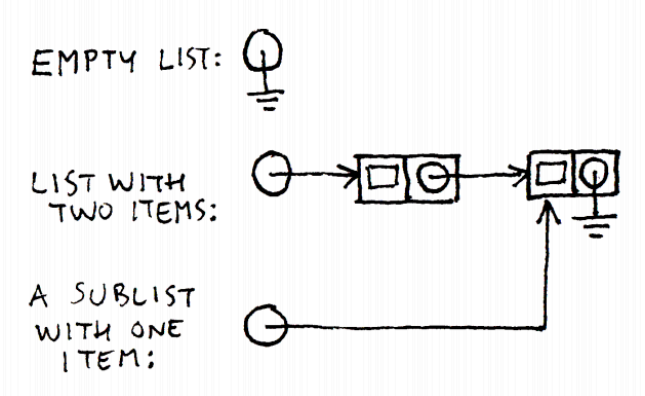
We also define the fact that we have the ability to declare record data types – collection of elements (fields), possibly of different types, accessed using a name – **struct / class**

List ADT

1. Boolean isEmpty()
2. Item head()
3. Void prepend(item x)
4. List tail()
5. Void setTail(List newTail)

```
0  ADT List {
1    boolean isEmpty();
2    // BEHAVIOUR: Return true iff the structure is empty.
3
4    item head();
5    // PRECONDITION: isEmpty() == false
6    // BEHAVIOUR: return the first element of the list (without removing it).
7
8    void prepend(item x);
9    // BEHAVIOUR: add element <x> to the beginning of the list.
10   // POSTCONDITION: isEmpty() == false
11   // POSTCONDITION: head() == x
12
13   List tail();
14   // PRECONDITION: isEmpty() == false
15   // BEHAVIOUR: return the list of all the elements except the first (without
16   // removing it).
17
18   void setTail(List newTail);
19   // PRECONDITION: isEmpty() == false
20   // BEHAVIOUR: replace the tail of this list with <newTail>.
21 }
```

In a singly linked list, you have the following structure where each item has a value and a pointer to the rest of the list (the next item in it).



In practise, there are likely to be overheads with this approach that would be better to avoid, for example overhead in creation storage and creation. Therefore you could implement the list using a large array, where an array element would be a pair of {payload, nextIndex}.

In some languages, like C, you can go even further. If the pointer and payload are the same size, then we can have an array of primitive type such that the payload and pointer fit into consecutive slots. Now, we can also get rid of the pointers which are simply pointing to the next memory location, and create a Boolean flag with the flag that the next item in the list being a pointer to the rest of the list in a different location.

Vector ADT

Vector ADT abstracts the notion of an array, assigning elements rank and allowing direct access to them:

1. Item elemAtRank(r)
2. void insertAtRank(r, item o)
3. void replaceAtRank(r, item o)
4. void removeAtRank(r)

```
0  ADT List {
1    item elemAtRank(r);
2    // BEHAVIOUR: Return the element at r
3
4    item insertAtRank(r,item o);
5    // PRECONDITION: vector rank is >= r
6    // BEHAVIOUR: insert item o at rank r
7
8    item replaceAtRank(r,item o);
9    // PRECONDITION: vector rank is >= r
10   // BEHAVIOUR: replace element with rank r with o
11
12   item removeAtRank(r);
13   // PRECONDITION: vector rank is >= r
14   // POSTCONDITION: vector size reduced by 1
15   // BEHAVIOUR: remove the item at rank r
16 }
```

Clearly, arrays are an obvious candidate for a vector datastructure, but it requires a mutable size, which copying the entire array. You could also use a linked list as the data structure, but this would give inferior performance for access but marginally better insertion / removal.

Stack ADT

This is a LIFO (Last In First Out) structure.

1. Boolean isEmpty()
2. Void push(item x)
3. Item pop()
4. Item top()

```
ADT Stack {
  boolean isEmpty();
  // BEHAVIOUR: return true iff the structure is empty.

  void push(item x);
  // BEHAVIOUR: add element <x> to the top of the stack.
  // POSTCONDITION: isEmpty() == false.
  // POSTCONDITION: top() == x

  item pop();
  // PRECONDITION: isEmpty() == false.
  // BEHAVIOUR: return the element on top of the stack.
  // As a side effect, remove it from the stack.


  item top();
  // PRECONDITION: isEmpty() == false.
  // BEHAVIOUR: Return the element on top of the stack (without removing it).
}
```

The stack ADT given above does not make any allowance for the push operation to fail (StackOverflow), though on any real computer with finite memory it must be possible to do enough successive pushes to exhaust some resource.

There are two often used implementations of the Stack datatype:

1. Array implementation
    a. Use an integer as the index as the top of the stack.
    b. The push operation writes a value into the array and increments the index, while pop does the converse.
2. Linked List implementation
    a. Pushing just adds an extra cell to the front of a list
    b. Popping removes it
    c. Both work by modifying stacks in place, so original stack is not usable.

Uses of Stack

1. PostScript
2. Reverse Polish Notation

Queue ADT

This is a FIFO (First in First out) structure

1. Boolean isEmpty()
2. Void put(item x)
3. Item get()
4. Item first()

```
ADT Queue {
  boolean isEmpty();
  // BEHAVIOUR: return true iff the structure is empty.

  void put(item x);
  // BEHAVIOUR: insert element <x> at the end of the queue.
  // POSTCONDITION: isEmpty() == false

  item get();
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the first element of the queue, removing it
  // from the queue.

  item first();
  // PRECONDITION: isEmpty() == false
  // BEHAVIOUR: return the first element of the queue, without removing it.


}
```

Deque (Double-ended queue): This is a variant of a queue and is accessible from both ends both for insertions and extractions.

```
ADT Deque {
  boolean isEmpty();

  void putFront(item x);
  void putRear(item x);
  // POSTCONDITION for both: isEmpty() == false

  item getFront();
  item getRear();
  // PRECONDITION for both: isEmpty() == false
}
```

Stacks and Queues are effectively restricted Dequeues in which only one put and one get are enabled.

Set and Dictionary ADT

Sets

A set is a collection of **distinct** objects.

A static set will have the following:

```
  boolean isEmpty()
  // BEHAVIOUR: return true iff the structure is empty.

  boolean hasKey(Key x);
  // BEHAVIOUR: return true iff the set contains a pair keyed by <x>.

  Key chooseAny();
  // PRECONDITION: isEmpty() == false
```

```
8   // BEHAVIOUR: Return the key of an arbitrary item from the set.
9
10  int size();
11  // BEHAVIOUR: Return the cardinality (size) of the set
```

A dynamic set will add in the ability to add and remove elements as well as add common mathematical operations:

```
0   Set union (Set s, Set t);
1   // BEHAVIOUR: return the union of sets s and t
2
3   Set intersection(Set s, Set t);
4   // BEHAVIOUR: return the intersection of sets s and t
5
6   Set difference(Set s, Set t);
7   // BEHAVIOUR: Return the difference of sets s and t
8
9   boolean subset(Set s, Set t);
10  // BEHAVIOUR: Return whether set s is a subset of set t
```

It is important to note that the ADT does not say anything about whether the operations are destructive – whether you come back with a modified version of inputs or a new Set altogether.

In order to add an order to the elements, we could also add the following:

```
0   Key min();
1   // PRECONDITION: isEmpty() == false
2   // BEHAVIOUR: Return the smallest key in the set.
3
4   Key max();
5   // PRECONDITION: isEmpty() == false
6   // BEHAVIOUR: Return the largest key in the set.
7
8   Key predecessor(Key k);
9   // PRECONDITION: hasKey(k) == true
10  // PRECONDITION: min() != k
11  // BEHAVIOUR: Return the largest key in the set that is smaller than <k>.
12
13  Key successor(Key k);
14  // PRECONDITION: hasKey(k) == true
15  // PRECONDITION: max() != k
16  // BEHAVIOUR: Return the smallest key in the set that is larger than <k>.
```

Dictionary

A dictionary associates keys with values and allows you to look up the relevant value if you supply the key. They are also known as Maps in Java. In a dictionary, the mapping between keys and values is a function, ie you can't have different values associated with the same key. Our ADT is:

```
ADT Dictionary {
  void set(Key k, Value v);
  // BEHAVIOUR: store the given (<k>, <v>) pair in the dictionary.
  // If a pair with the same <k> had already been stored, the old
  // value is overwritten and lost.
  // POSTCONDITION: get(k) == v

  Value get(Key k);
  // PRECONDITION: a pair with the sought key <k> is in the dictionary.
  // BEHAVIOUR: return the value associated with the supplied <k>,
  // without removing it from the dictionary.

  void delete(Key k);
  // PRECONDITION: a pair with the given key <k> has already been inserted.
  // BEHAVIOUR: remove from the dictionary the key-value pair indexed by
  // the given <k>.
}
```

The ADT does not provide a way of asking if a key is in use and does not mention anything about ensuring that a dictionary does not get overcrowded and mention anything about max fill of a dictionary, both of which need to be tackled in a practical implementation.

Data Structures for Sets / Dictionaries

Vector Representation

When the keys are known to be drawn from the set of integers in the range 0…n (**for some tractable n),** the dictionary can be modelled directly by a simple vector, and both set() and get() operations have unit cost. If the key values come from some other integer range, then we can simply subtract the minimum value.

This strategy is known as **Direct Addressing** and if the number of keys used is much smaller than n, direct addressing becomes extremely inefficient even though it remains O(1) in time performance.

Lists

For sparse sets, you could use a list, with each item a record which stores a key-value pair. The get() function would scan along the list, searching for the desired key. The set() function could either place it at the start of the list or search for an existing item with the same key (to update), before placing it at the end. In the second case, duplicate keys are avoided and so it would be legal to make arbitrary permutations of the list.

Get() and set() are clearly O(n) with get() taking n/2 searches on average and set() always taking n (for a new item to be added).

Arrays

We could clearly use an array instead of a list in a similar way. However, we need to be careful about the size of the array, as we would need to copy to a new array every time the array needs to be increased in size (amortized O(1) if we double every time).

Retrieval can however be done using a binary search into the array – therefore it is Θ(lg n) rather than n for the list.

Binary Search Trees

A BST holds a number of keys within a tree structure to help with searching. The keys must have a total order. Each node of the binary tree will contain an item (consisting of a key-value pair) and pointers to two sub trees, the left one for all items with keys smaller than the node's one and the right one for all items with larger keys.

Searching: Compare the sought key with the visited node and, until you find a match, recurse down the left or right subtree as appropriate – therefore O(height of tree H)

Insertion: Follow the search procedure, moving left or right as appropriate until you discover the key already exists in the tree or you reach a leaf. Update / Insert the tree as appropriate – therefore cost of O(H)

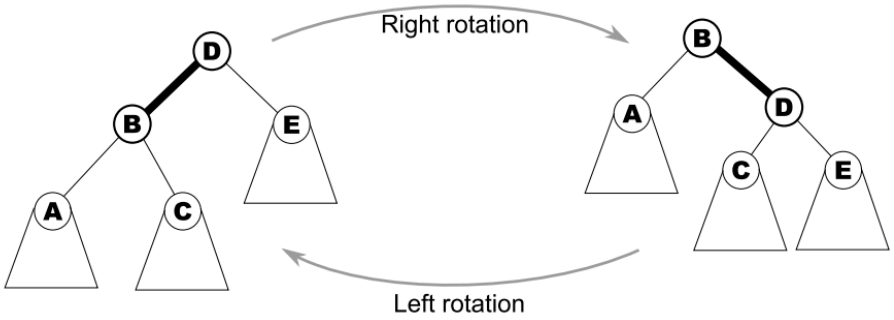
Deletion: Search to find the key. If it's a leaf, deletion is trivial. If not, there are various options:

1. A non-leaf with only one child can simply be replaced with its child
2. For a non-leaf with two children, replace it with its successor – then item can't have two children and can thus be deleted in one of the ways already seen.
    a. Meanwhile, newly moved up object satisfies the order requirements that keep the tree structure valid

This is clearly O(H)

Successor: If the given node has a right subtree then the successor is in it – go right and then go left as far as possible. If there is no right subtree, then go up until you can go right – this is the successor. If you reach the node, then the root has no successor.

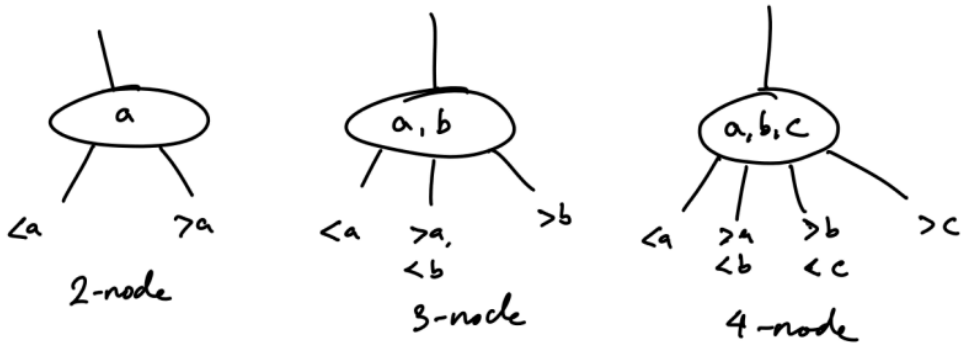
Rotation: Sometimes can be useful to rotate edges around nodes in a BST. A rotation is a local transformation of a BST that changes the shape of the BST but preserves the BST properties.



The general problem with trees is getting them balanced. In a perfectly balanced tree, each branch will be $\lceil \lg n \rceil$ but in the worst case, we will get a linear list of nodes with corresponding linear performance. While we could intelligently apply rotation to make a tree as balanced as possible on paper, this is hard to do with a computer.

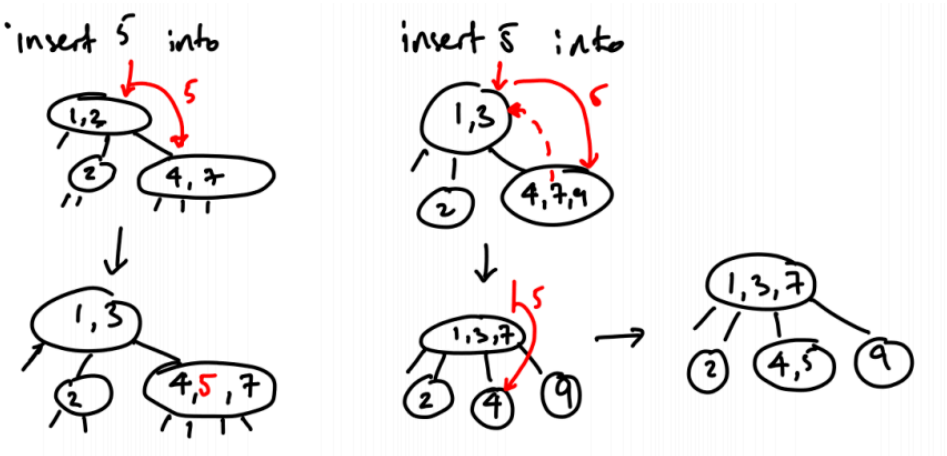
2-3-4 Trees

2-3-4 trees are composed of three types of nodes – a 2-node, 3-node and a 4-node:



Insertion:

1. Walk down the tree following the edges that contain the key to insert until we get to a leaf node
2. If the leaf node is a 2 or 3 node, insert into that node and finish
3. Else if the leaf node is a 4-node (with say keys a, b, c), we first explode it. The middle element c is moved up a level in the tree inserting into the parent node or making a new node if there is no parent. If we do insert c in the parent and it was already a 4-node, we have to explode that node and so on up the tree.
4. Continue out walk to the leaf node (now guaranteed not to be a 4-node) and insert into it.



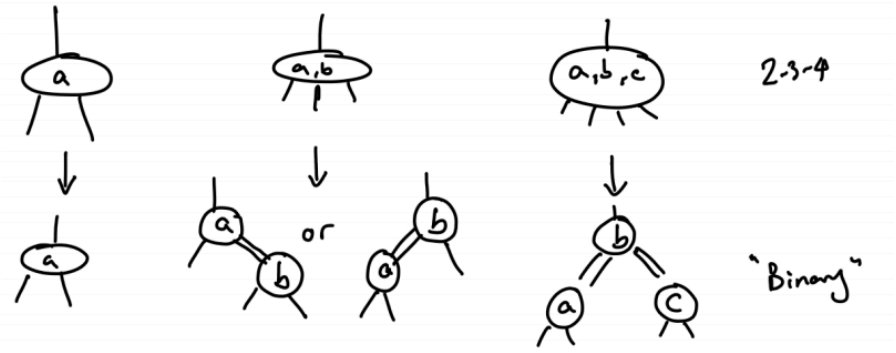
The key to a 2-3-4 tree is that the only way you can increase the length of a branch is to explode the root, i.e. we expand at the top and not the bottom as per BSTs. This means that when you increase the length of one branch by one, all the other branches increase by one two. The tree remains triangular (balanced)! However, it is clearly very hard to represent in code, with it being space efficient and performing well.

Important Subtlety: *While the branch lengths are the same, the cost of traversing them are not necessarily the same, since they can have 2-nodes (1 comparison), 3-nodes (2 comparisons), 4-node (2 comparisons)*

Red-Black Trees

In a Red-Black tree, we map our 2-3-4 tree as a binary tree.

- A 2-node is easy – it's already a binary node
- A 3-node has three outgoing node and a 4-node has 4 which can't be made from a single binary node, but we can glue binary nodes together to get what we want.



Connections of items of the same node in a 2-3-4 tree can be represented as a red line or a double black line.

Insertion

1. Add a new key using BST rules and adding it as a red node
2. If this gets red followed by red, try and rotate out of it (if we don't have a 4 node)
3. If rotations can't help, explode by pushing up red edges one level (and maybe beyond that).

RULES

1. Every node is either red or black
    a. Of course, it is – it has either been glued to another node or it hasn't
2. Root is black
    a. The root has no incoming link (or it would not be the root). So it must be black.
3. Leaves are black and never contain key-value pairs
    a. We took this for granted in the 2-3-4 tree and it applies to the red-black as well
4. If a leaf is red, both its children are black
    a. We can't get a glue edge followed by a glue edge. This is because no 2-3-4 node with these properties – and nodes are always connected with black linked.
5. For each node, all paths from that node to descendant leaves contain the same number of black nodes
    a. A black link corresponds to a link in the 2-3-4 tree. All paths in 2-3-4 have same number of nodes and therefore same number of black links.

Analysis

All accesses are bound by the black height. If the tree were purely black we would have:

$$n = 2^{hb} - 1$$

Adding in red nodes doesn't change the black height, so we know:

$$n \geq 2^{hb} - 1 \Rightarrow hb \leq \lg(n+1)$$

In the worst case, there is one red node for every black node – black always follows red therefore

$$hb = \frac{height}{2} \Rightarrow h \leq 2\lg n + 1 = O(\lg n)$$

Therefore, performance is O(lg n) for everything:

	Average	Worst Case
Insert	O(lg n)	O(lg n)
Delete	O(lg n)	O(lg n)
Search	O(lg n)	O(lg n)

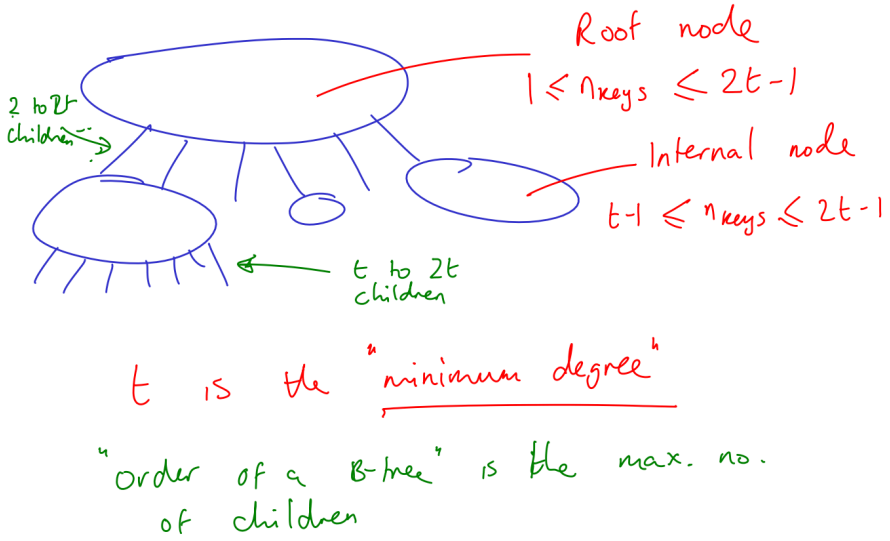
B-trees

So far, we have assumed that the data just sits in RAM, but if it's too big, we could just use a hard disk. But that has sequential access and a BST isn't a great choice, as we would access each node in one cycle for the I/O and every time you follow a link you need to read the next node from the disc. This becomes disc-bound and is therefore very slow.

2-3-4 trees were better with this regard, having fewer levels by sticking multiple keys together in bigger nodes. Big nodes work particularly well for us now, as we can write and read big nodes in one operation. B-trees extends the idea to much wider trees.

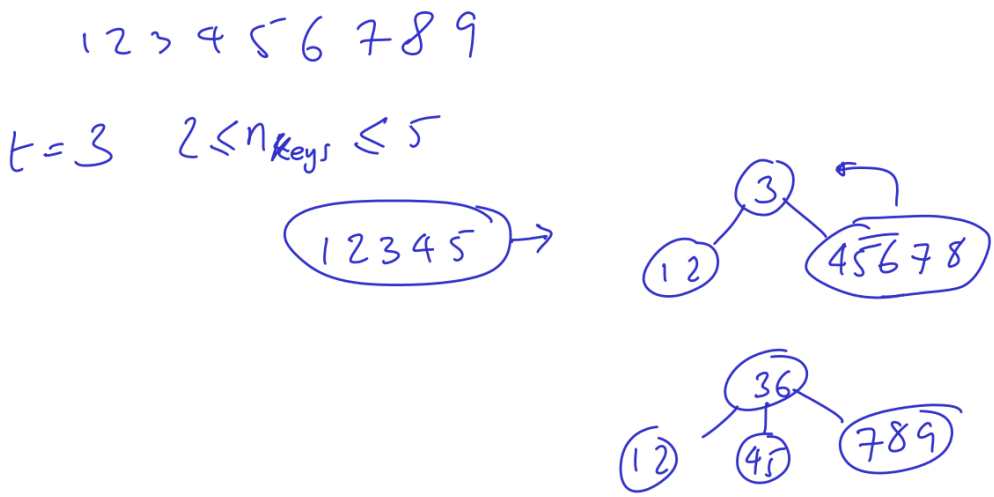
Rules

1. There are internal nodes (with keys and payloads and children) and leaf nodes (without keys or payloads or children).
2. For each key in a node, the node also holds the associated payload.
3. All lead nodes are at the same distance form the root.
4. All internal nodes have at most 2t children; all internal nodes except the root have at least t children (**t is referred to as the minimum degree – 2-3-4 has minimum degree of 2)**
5. A node has c children iff it has c-1 keys



The minimum degree is the minimum number of children (NOT KEYS). Knuth prefers the idea of order (max number of children). This is more flexible as it allows us to have a maximum degree that is odd.

Insertion: The same idea as 2-3-4 except we split nodes into two based on the median being pushed up.



This is clearly **O(h)**

Searching: Searching is done exactly the same as for the 2-3-4 tree and is clearly O(h) as the cost is traversing the links.

Deletion: Deletion is more elaborate as it involves numerous subcases. You can't delete a key from anywhere other than a bottom node, otherwise you upset its left and right children that lose their separator. In addition, you can't delete a key from a node that already has the minimum number of keys. So, the general algorithm consists of creating the right conditions and then deleting.

To move a key to a bottom node from the purposes of deleting it, swap it with its.

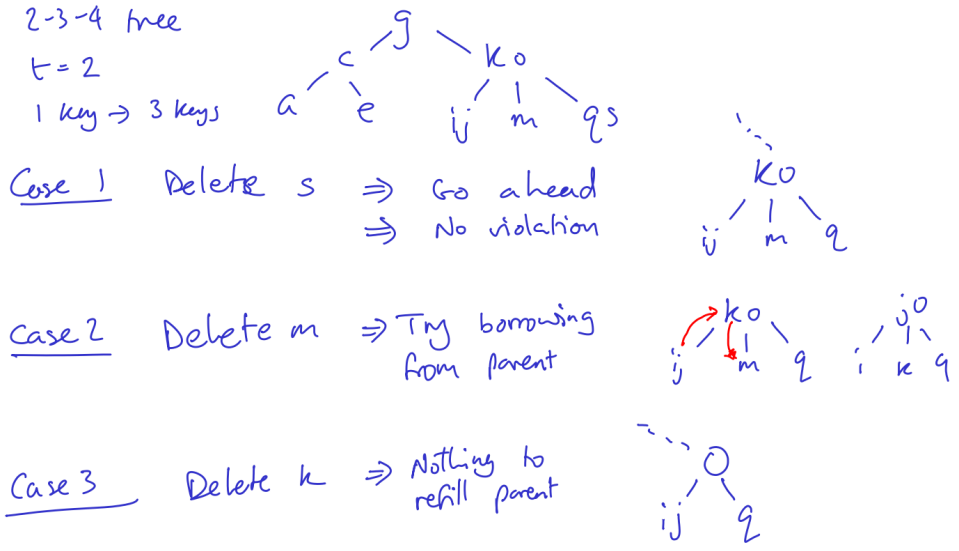
To refill a node that has too few keys, use an appropriate combination of the following three operations, which rearrange a local part of a B-tree in constant time preserving the B-tree operations:

1. **Merge**
    a. Merge two adjacent brother nodes and the key that separates them from the parent node. The parent node loses one key.
2. **Split**
    a. The reverse operations splits a node into three: a left brother, a separating key and a right brother. The separating key is sent up to the parent.
3. **Redistribute**
    a. Redistributes the keys among two adjacent sibling nodes. Merge followed by a split in a different place (this place being the centre of the large merged node).

Each of these operations is only allowed if the new nodes respect their min and max capacity constraints, therefore this is the pseudocode:

```
def delete(k):
    """B-tree method for deleting key k.
    PRECONDITION: k is in this B-tree.
    POSTCONDITION: k is no longer in this B-tree."""

    if k is in a bottom node B:
        if B can lose a key without becoming too small:
            delete k from B locally
        else:
            refill B (see below)
            delete k from B locally
    else:
        swap k with its successor
        # ASSERT: now k is in a bottom node
        delete k from the bottom node with a recursive invocation
def refill(B):
    """B-tree method for refilling node B.
    PRECONDITION: B is an internal node of this B-tree, with t-1 keys.
    POSTCONDITION: B now has more than t-1 keys."""

    if either the left or right sibling of B can afford to lose any keys:
        redistribute keys between B and that sibling
    else:
        # ASSERT: B and its siblings all have the min number of keys, t-1
        merge B with either of its siblings
        # ...this may require recursively refilling the parent of B,
        # because it will lose a key during the merge.
```

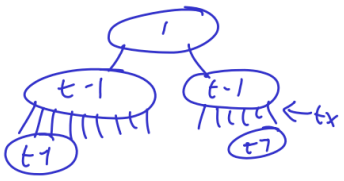
EXAMPLE:



Bounding the B-Tree Height

n keys; min degree t; height H

In the worst case, we are trying to make H as large as possible, while minimising the keys for the notional H.



d	#Nodes	# keys
1	1	1
2	2	$2(t-1)$
3	$2(t)$	$2t(t-1)$
4	$2t^2$	$2t^2(t-1)$
⋮		
H	$2t^{H-2}$	$2t^{H-2}(t-1)$

$$Nmin = 1 + 2(t-1) + 2t(t-1) + \cdots + 2^{H-2}(t-1) = 1 + \frac{2(t-1)(t^{H-1}-1)}{t-1}$$
$$= 1 + 2(t^{H-1} - 1)$$
$$N \geq 2t^{H-1} - 1 \Rightarrow H \leq 1 + \log_t \frac{n+1}{2}$$
$$H = O(\lg n)$$

Hash Tables

A hash table implements the general case of the Dictionary ADT where keys do not have a total order defined on them. In fact, even when the keys used to have an order relationship associated with them, it may be worth looking for a way of building a dictionary without using this order. **Binary search makes locating items in a dictionary easier by imposing a coherent and ordered structure; hashing, instead, places its bet the other way, on chaos.**

A hash function maps a key onto an integer between 0 and some maximum, and for a good hash function, this mapping will appear to have hardly any pattern.

There can be collisions, where two keys map to the same value, in this case there are two main strategies for handling it, **chaining and open addressing.**

Chaining: We arrange that the locations of the array hold linear lists that collect all the items that hash to that particular values. This should lead to lists of average length n/m (where n is the number of keys and m the size of the table being used)

Performance:

1. Insertion – O(1) to compute hash and O(1) to add to the slot and add to the front / back of the list – therefore O(1)
2. Deletion, in the best case, it is O(1), in the worst case it is O(n)
3. Failed Search
    a. Worst case it is O(n)
    b. In the average case, the average list length is n/m and therefore the cost is O(1 + n/m)
4. Successful search
    a. Worst case – O(n)
    b. Average case – assume uniform hasing
        i. Consider the ith insertion

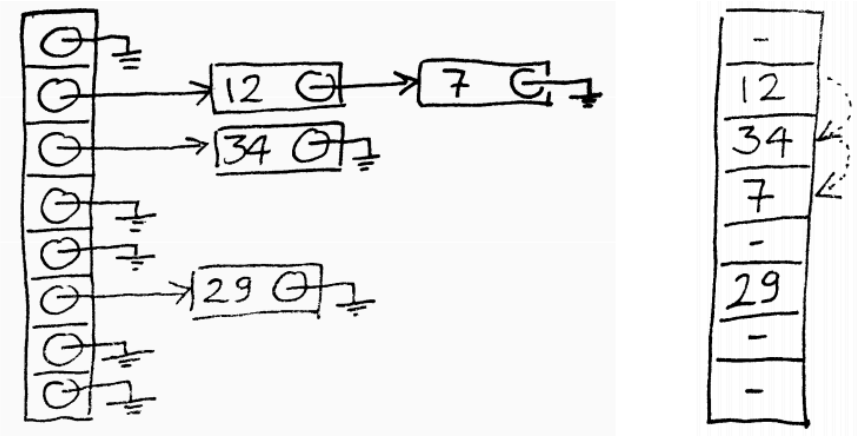
ii. Added to a list of length (i-1)/m
iii. To find i, costs 1 + (i-1)/m
iv. Average of all i
1. $Cost = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = \frac{1}{n}\left(n+\frac{1}{m}\sum i - \frac{n}{m}\right)$
2. $= 1 + \frac{n+1}{2m} - \frac{2}{2m} = 1 + \frac{n+1}{m} - \frac{1}{m} = O(1+\alpha)$

Constant?
As long as we keep n/m constant (the load factor), we can consider the costs to be constant.

Open Addressing: Ideas is that h(key) is just a first preference for where to store the given key in the array. On adding a new key, if that location is empty then it can be used, otherwise a succession of other probes is made of the hash table according to some rule until the key is found to be present or an empty slot for it is located. The simplest method is to try successive array locations from the place of the first probe, wrapping around at the start of the array.

It is important to note that the performance of a hash table decreases significantly when the hash table gets nearly full and implementations will typically double the size of the table once occupancy goes above a certain threshold.

Diagram of Chaining (left) vs Open Addressing (right)



Performance
Worst case cost of using a hash table is bad – if all values hash to the same value. Average case is very good as long as the number of values hashed is much lower than the size of the hashtable, then it is constant cost

PROBING SEQUENCES FOR OPEN ADDRESSING
There are many strategies to determine the sequence of slots to visit until a free one is found. In order to not waste slots, we always want to have a sequence that will visit every slot before revisiting any (should never do this and end before this).

```
0 int probe(Key k, int j);
1 // BEHAVIOUR: return the array index to be probed at attempt <j>
2 // for key <k>.
```

```
Int m = size of hash table;
```

Linear Probing: Just returns h(k) + j mod m. Therefore always try the next cell in sequence. This is simple to understand and implement, but leads to **primary clustering**, where many failed attempts hit the same slot and spill over to the same follow-up spots. The result is longer and longer runs of occupied slots, increasing search time.

Quadratic Probing: With quadratic probing, you return h(k) + cj + dj^2 mod m for some constants c and d. This works much better than linear probing, provided that c and d are chosen appropriately: when two distinct probing sequences hit the same slot, in subsequent probes they then hit secondary slots. However, still leads to **secondary clustering** because any keys that hash to the same value will yield the same probing sequence.

Double Hashing: Probing sequence is $h_1(k) + h_2(k)$ mod m, using two different hash functions h_1 and h_2. Even keys that hash to the same value (under h1) are assigned different probing sequences. However, access costs another hash function computation.

Using the probing sequence

1. GET
    a. Correct slot is the first one in the sequence which contains the sought key and if an empty slot is found then the key is not in the dictionary
    b. A 'deleted' tag should be passed through
    c. Otherwise, if m probes completed, then not in hash table
2. SET
    a. If a slot with the sought key is found, then that is the one to use.
    b. Otherwise, the first empty slot (can use slots with a deleted tag) can be used.
    c. If m probes unsuccessful, array full.
3. DELETE
    a. Find the slot to delete using GET
    b. Mark it as deleted
        i. This solution makes the GET operation become slower and slower if lots of deletes are occurring.
        ii. Therefore we should rehash reasonably regularly after lots of deletes with not enough sets.

Open Addressing Search Performance

1. Average Number of Probes in a failed search
    a. $< \frac{1}{1-\alpha}$
2. Average Number of Probes in a successful search
    a. $\frac{1}{\alpha} ln\left(\frac{1}{1-\alpha}\right) + \frac{1}{\alpha}$

Rehashing: Create a new array (normally twice the size to amortize the cost of the operation), with a new hash function. Insert every key-value pair of the old array into the new one and

delete the old array. The deleted slots are not copied. **This is necessary when the load factor becomes too high – HashMap in Java has a rehash load factor of 75%.**

Chaining vs Open Addressing

- **Open Addressing**
  - Faster for low load factor: slow as load factor approaches 1
  - Better cache performance
  - Good hash functions hard to find
  - Best for small records that fit completely in the array and fit in a cache line
- **Chaining**
  - Can keep growing beyond a load factor of 1 (at cost of reduced performance)
  - Has extra space requirements.

Why don't we always use a hash-table?

- To keep the load factor low, we need large arrays – not space efficient
- Worst case is O(n), much worse than RB tree at O(lg n)
- O(1) assumes a uniform distribution of keys but this isn't guaranteed
- We lose all notion of order and can't iterate over the keys in a meaningful way

Graph Algorithms

Notation and Representation

A graph is a set of vertices (or nodes, or locations) and edges (or connections) between them.

- A graph can be denoted by g = (V, E) where V is the set of vertices and E is the set of edges.
- A graph may be directed or undirected.
  - For a directed graph v1 -> v2 denotes an edge from v1 to v2
  - For an undirected graph, v1 – v2 denotes an edge from v1 to v2 and from v2 to v1
- We assume there are not multiple edges between a pair of nodes
- A path in a graph is a sequence of vertices connected by edges
  - They may visit the same vertex more than once
- A cycle is a path from a vertex back to itself
- A **directed acyclic graph** or DAG is a directed graph without any cycles.
- An undirected graph is **connected** if for every pair of vertices there is path between them. A **forest** is an undirected acyclic graph.
- **A tree is a connected forest.**

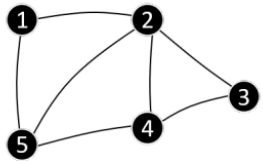
There are generally two ways to store a graph in computer code: as an Adjacency List or as an Adjacency Matrix

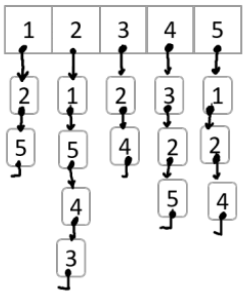
Adjacency List

Takes up O(V + E) space, and here we store (for each vertex) a list of all the points it can get to.

Adjacency Matrix

Takes up $O(V^2)$ space, with a Boolean storage of whether we can reach a point from another point for every possible point. It lends itself to being easily and painlessly being extended to weighted graphs, where there is a weight to each of the paths.





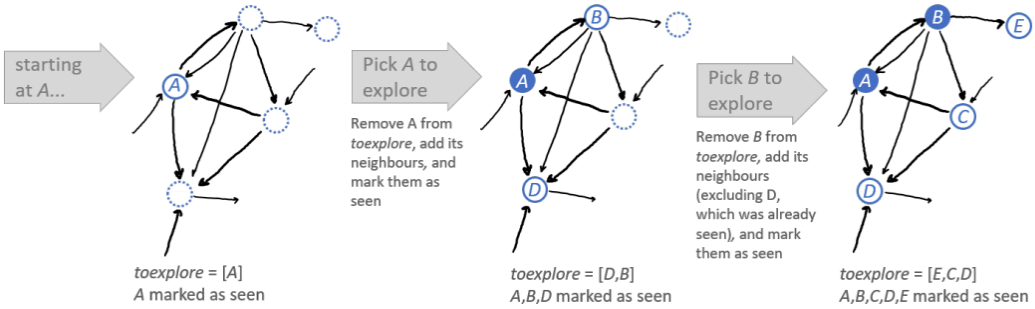
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

The choice of whether to use an adjacency list or matrix should depend on a few things (matrix is easier to read from (and quicker), but harder to form initially), but largely should be dependent on the density of the graph = E/V^2.

Breadth-First Search

In a breadth-first search the general idea is that you visit all nodes, starting at a particular node and visiting them as you see a path to them (and only visiting a node once).

In order to do this, we visit vertices, look at all of its neighbours and mark them as worth exploring if they have not been explored yet – with a 'Seen' flag for each vertex.



It is generally easiest to implement a Breadth-First Search using a Queue to store the list of vertices waiting to be explored.

```
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Queue([s])  # a Queue initially containing a single element
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popright()   # Now visiting vertex v
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushleft(w)
                w.seen = True
```

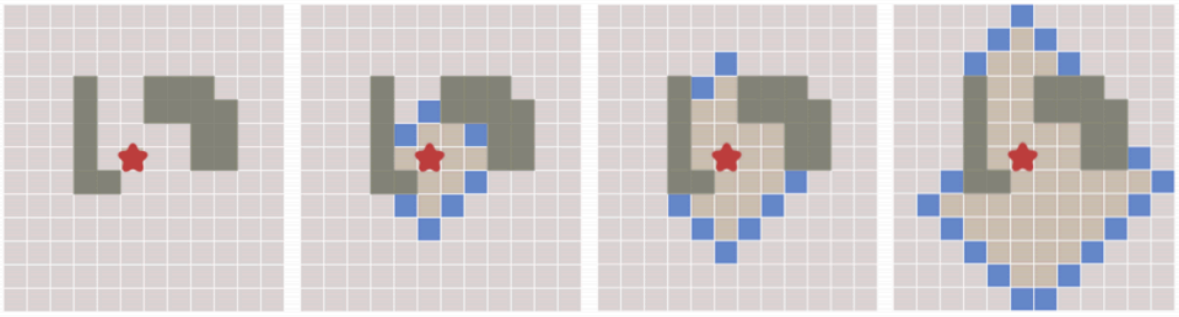
With a small tweak, we can also adapt this code to find the shortest path between a pair of nodes. All it takes is keeping track of how we discovered each vertex.

```
# Find a path from s to t, if one exists
def bfs_path(g, s, t):
    for v in g.vertices:
        v.seen = False
        v.come_from = None
    s.seen = True
    toexplore = Queue([s])

    # Traverse the graph, visiting everything reachable from s
    while not toexplore.is_empty():
        v = toexplore.popright()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushleft(w)
                w.seen = True
                w.come_from = v

    # Reconstruct the full path from s to t, working backwards
    if t.come_from is None:
        return None  # there is no path from s to t
    else:
        path = [t]
        while path[0].come_from != s:
            path.prepend(path[0].come_from)
        path.prepend(s)
        return path
```

Analysis

The initialisation is run for every vertex, so it is O(V). The main exploration bit is run, at most O(V) times, since we check the seen flag to ensure that each vertex enters the queue at most once. Finally, the line which checks whether it is seen is run for every edge at most once, or twice for an undirected graph, therefore it is O(V + E)

The reason that breadth first search is effective is because it will always find the best solution first. This is because, with every iteration of going through the queue it is exploring a frontier and moving outwards, moving out from all possible paths before continuing on the first increased path, as per this diagram it moves out in a circular area.



Depth-First Search

Depth-First Search is a backtracking based algorithm, where you attempt to go as far down a path as possible before backtracking if you hit a dead end then take a different route, backtracking as little as possible.

We can use a stack to store all the vertices waiting to be explored.

```
1  # Visit all vertices reachable from s
2  def dfs(g, s):
3      for v in g.vertices:
4          v.seen = False
5      toexplore = Stack([s])   # a Stack initially containing a single element
6      s.seen = True
7
8      while not toexplore.is_empty():
9          v = toexplore.popright()   # Now visiting vertex v
10         for w in v.neighbours:
11             if not w.seen:
12                 toexplore.pushright(w)
13                 w.seen = True
```

It is also possible to do this using recursion, as here:

```
1  # Visit all vertices reachable from s
2  def dfs_recurse(g, s):
3      for v in g.vertices:
4          v.visited = False
5      visit(s)
6
7  def visit(v):
8      v.visited = True
9      for w in v.neighbours:
10         if not w.visited:
11             visit(w)
```

It is clear that the dfs algorithm has O(V+E) running time based on exactly the same logic as the analysis used for the breadth first search algorithm.

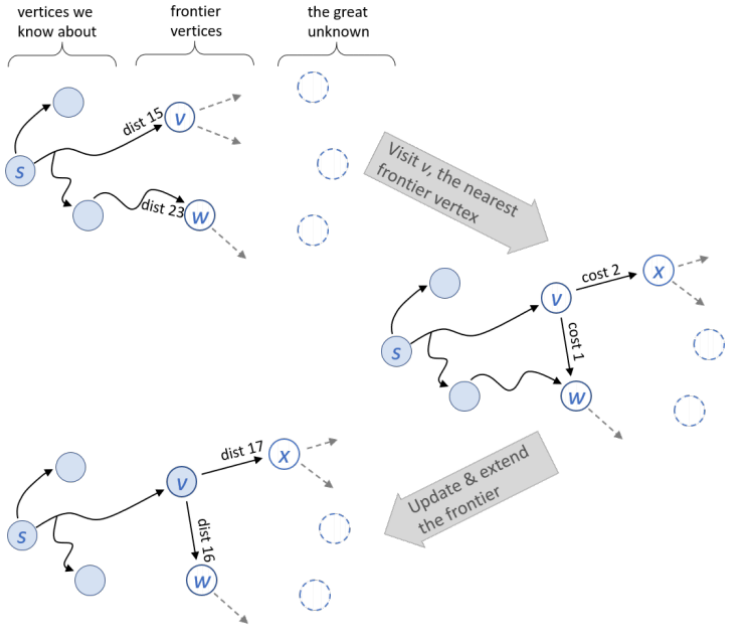
The recursive equivalent is clearly also the same, hence: Line 4 is called once per vertex; visit is called once per vertex (when we check if it is visited) and the for loop is gone through once per edge in total over the entire process, therefore it O(V + V + E) = O(V + E)

Dijkstra's

Problem Statement: Given a directed graph where each edge is labelled with a cost ≥ 0, and a start vertex s, compute the distance from s to every other vertex

Dijkstra's algorithm gets the shortest path between two nodes in a weighted graph. In a breadth first search, we visited the vertices in order of how many hops they are from the start vertex. For Dijkstra's algorithm, we are visiting them in order of distance from the start vertex.

By keeping track of a frontier of vertices we haven't explored, we have a queue to visit, organised by distance (priority queue!)




```
1  def dijkstra(g, s):
2      for v in g.vertices:
3          v.distance = ∞
4      s.distance = 0
5      toexplore = PriorityQueue([s], sortkey = lambda v: v.distance)
6
7      while not toexplore.isempty():
8          v = toexplore.popmin()
9          # Assert: v.distance is the true shortest distance from s to v
10         # Assert: v is never put back into toexplore
11         for (w, edgecost) in v.neighbours:
12             dist_w = v.distance + edgecost
13             if dist_w < w.distance:
14                 w.distance = dist_w
15                 if w in toexplore:
16                     toexplore.decreasekey(w)
17                 else:
18                     toexplore.push(w)
```

In fact, given the assertion on line 10, we could insert all nodes into the queue in line 5 and get rid of lines 15, 17 and 18.

Correctness

Theorem: The algorithm terminates. When it does, for every vertex v, the value v.distance it has computer is equal to the true distance from s to v. Furthermore, the two assertions are true.

Proof of Assertion 9: Suppose the assertion fails, let v be the vertex for which first fails. Consider a shortest path from s to v:

s = u1 -> … -> uk = v

Let u_i be the first vertex in this sequence which has not been popped from toexplore so far at this point in execution. Then:

- distance(s to v)
  - < v.distance
    - Since assertion failed
  - ≤ u_i.distance
    - Since toexplore is a Priority Queue which had both u_i and v
  - ≤ u_{i-1}.distance + cost(u_{i-1} -> u_i)
    - By lines 13-18 when u_{i-1} was popped
  - = distance(s to u_{i-1}) + cost(u_{i-1} -> u_i)
    - Assertion didn't fail on u_{i-1}
  - ≤ distance (s to v)

This is a contradiction, therefore the premise (that Assertion 9 failed at some point) is false.

Proof of Assertion 10

Once a vertex v has been popped, Assertion 9 guarantees that v.distance == distance (s to v). The only way that v could be pushed back into toexplore is if we found a shorter path to v which is impossible.

Since vertices can never be re-pushed into toexplore, the algorithm must terminate. At termination, all the vertices that are reachable from s must have been visited and popped, and when they were popped they passed Assertion 9. They can't have had v.distance changed subsequently.

Running Time

The exact running time depends on how the Priority Queue is implemented. If we use a Fibonacci Heap, we get O(1) (amortized) running time for both push() and decreasekey() and O(log n) for popmin(). Line 3 and 8 is run once per vertex, and line 12-18 are run once per edge. Therefore O(V + VlogV + E) = O(E + V log V)

Bellman-Ford

Problem Statement: Given a directed graph where each edge is labelled with a weight, and a start vertex s, (i) if the graph contains no negative-weight cycles reachable from s then for every vertex v compute the minimum weight from s to v; (ii) otherwise report that there is a negative weight cycle reachable from s.

Bellman-Ford can do effectively the same as Dijkstra, but while being able to deal with graphs with negative edge weights. The general idea is that relax all our edges (seeing if it could make us reach a node more easily) lots and lots of times. The effectiveness works in that we only have to do this a set number of times, the number of vertices in the graph.

```
1  def bf(g, s):
2      for v in g.vertices:
3          v.minweight = ∞ # best estimate so far of minweight from s to v
4      s.minweight = 0
5
6      repeat len(g.vertices)−1 times:
7          # relax all the edges
8          for (u,v,c) in g.edges:
9              v.minweight = min(u.minweight + c, v.minweight)
10             # Assert v.minweight >= true minimum weight from s to v
11
12     for (u,v,c) in g.edges:
13         if u.minweight + c < v.minweight:
14             throw "Negative−weight cycle detected"
```

Lines 8 to 12 iterate over all the edges, relaxing them and Line 6 iterates it the number of times according to the number of vertices there are. Then Lines 12-14 effectively say that if we get a different answer after V-1 rounds of relaxation and V rounds, then there is a negative-weight cycle and if we don't there is no negative weight cycle.

Performance

It is clear that the algorithm is O(VE + E) = O(VE)

Correctness

Theorem: The algorithm correctly solves the problem statement. In case (i) it terminates successfully, and in case (ii) it throws an exception in line 14. Furthermore, the assertion on line 10 is true

Proof of Assertion on line 10: Say w(v) is the true weight from s to v with the convention that w(v) = -∞ if there is a path that includes a negative weight cycle. The algorithm can only ever update v.minweight when there it has a valid path to v, therefore the assertion is true.

Proof for case (i): Pick any vertex v and consider a minimal-weight path from s to v. Let the path be s = u0 -> u1 -> … -> uk = v

Consider what happens in successive iterations of the main loop, lines 8-10

- Initially, s.minweight is correct
- After one iteration, u1.minweight is correct
    - If there was a lower weight path to u1, then the path we are considering wouldn't be the minimal-weight path to v.
- After two iterations, u2.minweight is correct, etc

Since there is no cycle (if there was any cycle, it would have weight > 0) so would not have the shortest path. The path has at most |V| – 1 edges, so after |V| - 1 iterations, v.minweight is correct. Therefore, by the time we reach line 12 all vertices have the correct minweight, so we terminate without exception

Proof for case (ii): Suppose there is a negative weight cycle reachable from s

$$s \to \cdots \to v_0 \to v_1 \to \cdots \to v_k \to v_0$$

where

$$weight(v_0 \to v_1) + \cdots + weight(v_k \to v_0) < 0$$

If the algorithm terminates without throwing an exception, then all the edges pass the test in line 13, i.e.

$$v_0.minweight + weight(v_0 \to v_1) \geq v_1.minweight$$
$$v_1.minweight + weight(v_1 \to v_2) \geq v_2.minweight$$

etc

$$v_k.minweight + weight(v_k \to v_0) \geq v_0.minweight$$

Therefore,

$$v_0.minweight + weight(v_0 \to v_1) + \cdots + weight(v_k \to v_0) \geq v_0.minweight$$

Therefore, the cycle has weight ≥ 0. This contradicts the premise – so at least one of the edges must fail the test in line 13 and so the exception will be thrown.

Johnson's Algorithm

Problem Statement: Given a directed graph where each edge is labelled with a weight, (i) if the graph contains no negative-weight cycles then for every pair of vertices compute the weight of the minimal weight path between those vertices; (ii) if the graph contains a negative-weight cycle then detect that this is so.

Allows us to compute shortest paths between all pairs of vertices.

The **betweeness centrality of an edge** is defined to be the number of shortest paths to use that edge over all the shortest paths between all pairs of vertices in a graph. It is a measure of how important an edge is, and its used for summarizing the shape of the graph.

A primary idea is that we could just run Dijkstra's algorithm V times, once from each vertex leading to a complexity of $O(VE + V^2 logV)$. If some edge weights are less than 0, we could run Bellman-Ford from each vertex, which would have running time $O(V^2E)$

But, we can Bellman-Ford once, then run Dijkstra once from each vertex, then run some cleanup for every pair of vertices, with running time = $O(VE) + O(VE + V^2 logV) + O(V^2) = O(VE + V^2 logV)$

This works by constructing an extra helper graph, running a computation in it and applying the results of the computation to the original problem. The helper graph is the original graph with an extra vertex s and zero weight edges s->v for all vertices v

Implementation:

1. **The Helper Graph:** Run Bellman-Ford on the helper graph and let the minimum weight from s to v to be dv. If Bellman-Ford reports a negative-weight cycle, then stop.

2. **The Tweaked Graph:** Define a tweaked graph which is like the original graph but with different edge weights
    a. $w'(u->v) = d_u + w(u->v) - d_v$
    b. In this graph, every edge has $w'(u->v) > 0$
        i. The relaxation equation, applied to the helper graph says that:
        ii. $d_v \leq d_u + w(u -> v)$ therefore $w'(u -> v) \geq 0$
3. **Dijkstra on the tweaked graph:** Run Dijkstra V times on the tweaked graph, once from each vertex. All edges have weight >= 0
    a. The claim is that the minimum-weight paths in the tweaked graphs are the same as in the original graph

Proof: Pick any two vertices p and q, and any path between them.
$p = v_0 -> v_1 -> \dots -> v_k = q$

$$\begin{aligned}\text{Weight in tweaked graph} &= d_p + w(v_0 -> v_1) - d_{v1} + d_{v1} + w(v1 -> v2) - d_{v2} + \dots \\ &= d_p + w(v0 -> v1) + w(v1 -> v2) + \dots + w(v(k-1) -> vk) - d_q \\ &= \text{weight in original graph} + d_p - d_q\end{aligned}$$

Since d_p-d_q is the same for every path from p to q, the ranking of the paths is the same in the tweaked graph as in the original graph.

4. **Wrap Up**
    a. $$\begin{matrix}\text{min weight} \\ \text{from } p \text{ to } q \\ \text{in original graph}\end{matrix} = \begin{matrix}\text{min weight} \\ \text{from } p \text{ to } q \\ \text{in tweaked graph}\end{matrix} - d_p + d_q$$

All-Pairs Shortest Paths with Matrices

There is another possible algorithm, which solves the same problem as for Johnson's, with a running time of $O(V^3 \log V)$ – worse than Johnson's but is very simple.

The general idea is utilising a matrix to solve the problem using dynamic programming:

> Let $M^{(\ell)}$ be a $V \times V$ matrix, where $M_{ij}^{(\ell)}$ is the minimum weight among all paths from i to j that have ℓ or fewer edges.

We can define M^l in terms of M^{l-1} and this leads to an algorithm for computing M^l. If we pick l big enough (at least the maximum number of edges in any path), then we've solved the problem.

We also define W as our path matrix (representing the paths between two nodes and their distance).

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight}(i \to j) & \text{if there is an edge } i \to j \\ \infty & \text{otherwise.} \end{cases}$$

This is a n x n matrix, where n is |W|

For each step, we can easily say that:

$$M_{ij}^{(\ell)} = M_{ij}^{(\ell-1)} \wedge \left[\left(M_{i1}^{(\ell-1)} + W_{1j} \right) \wedge \left(M_{i2}^{(\ell-1)} + W_{2j} \right) \wedge \cdots \wedge \left(M_{in}^{(\ell-1)} + W_{nj} \right) \right]$$
$$= \left(M_{i1}^{(\ell-1)} + W_{1j} \right) \wedge \left(M_{i2}^{(\ell-1)} + W_{2j} \right) \wedge \cdots \wedge \left(M_{in}^{(\ell-1)} + W_{nj} \right).$$

Where a ^ b means min(a, b). Therefore, this says to go from I to j in ≤ l hops, you could either get there in l-1 hops or you go from I to some other node in l-1 hops and then take the edge k -> j.

We can clearly spot this is the same as regular matrix multiplication except it uses addition instead of multiplication and minimum instead of addition (let's write this as ⊗)

```
1  Let M^(1) = W
2  Compute M^(V-1) and M^(V), using M^(ℓ) = M^(ℓ-1) ⊗ W
3  If M^(V-1) == M^(V):
4      return M^(V-1) # this matrix consists of minimum weights
5  else:
6      throw "negative weight cycle detected"
```

Correctness

We've shown the derivation process which shows that M_{ij}^{l} is the minimum weight among all paths ≤ l. The proof that lines 3-6 are correct is the same as for Bellman Ford (path can only be |V| - 1 long, so if anything changes between |V| - 1 and |V| then clearly there is a negative weight cycle.

Running time

As with Matrix Multiplication, it takes V^3 operations to compute ⊗, so the total time is $O(V^4)$. However, you can also reduce the time it takes to do it, by repeatedly squaring, rather than doing itself.

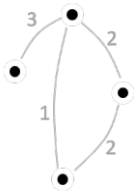
$$M^{(1)} = W$$
$$M^{(2)} = M^{(1)} \otimes M^{(1)}$$
$$M^{(4)} = M^{(2)} \otimes M^{(2)}$$
$$M^{(8)} = M^{(4)} \otimes M^{(4)}$$
$$M^{(16)} = M^{(8)} \otimes M^{(8)}$$
$$= M^{(9)} \quad \text{if there are no negative-weight cycles.}$$

Then, the running time is $O(V^3 \log V)$

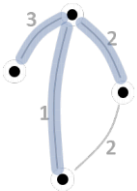
Prim's Algorithm

Problem Statement: Given a connected undirected graph with edge weights, construct an MST

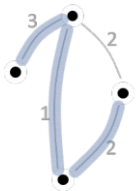
Given a connected undirected graph with edge weights, a minimum spanning tree is a tree that spans the graph (connects all the vertices, and has minimum weight among all spanning trees).



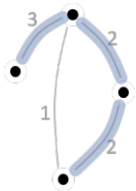
undirected graph with edge weights



spanning tree of weight 6

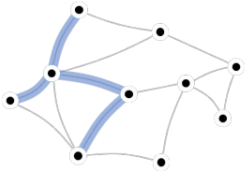


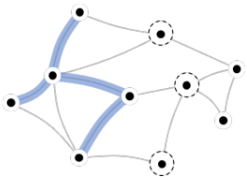
spanning tree of weight 6

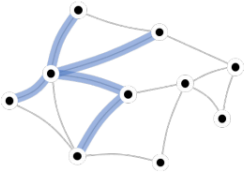


spanning tree of weight 7

Prim's Algorithm builds up the MST greedily, attempting (from a starting vertex chosen arbitrarily) to pick the cheapest edge that introduces a new edge.


a tree build up with four edges so far


three candidate vertices to add next


pick the cheapest of the four connecting edges and add it to the tree

```
1   def prim(g, s):
2       for v in g.vertices:
3           v.distance = ∞
4 +         v.in_tree = False
5 +     s.come_from = None
6       s.distance = 0
7       toexplore = PriorityQueue([s], lambda v : v.distance)
8
9       while not toexplore.isempty():
10          v = toexplore.popmin()
11 +        v.in_tree = True
12          # Let t be the graph made of vertices with in_tree=True,
13          # and edges {w—w.come_from, for w in g.vertices excluding s}.
14          # Assert: t is part of an MST for g
15          for (w, edgeweight) in v.neighbours:
16 ×            if (not w.in_tree) and edgeweight < w.distance:
17 ×                w.distance = edgeweight
18 +                w.come_from = v
19                  if w in toexplore:
20                      toexplore.decreasekey(w)
21                  else:
22                      toexplore.push(w)
```

The lines marked + are lines where we keep track of the edges which are in the tree and x are the modified lines from Dijkstra where we show that we are primarily only interested in distance from the tree rather than the distance from the start node.

Running Time

It is reasonably easy to see that Prim's algorithm terminates and it is nearly identical to Dijkstra's Algorithm and it therefore has the same running time = O(E + V log V) assuming the priority queue is implemented using a Fibonacci heap.

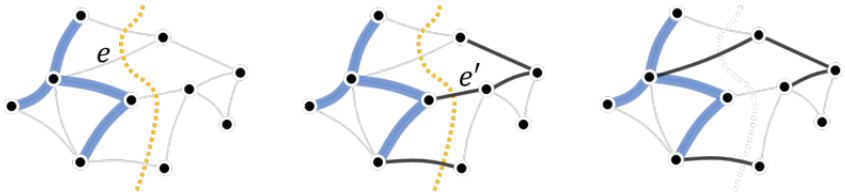
Correctness

In order to prove Prim's Algorithm does indeed find an MST, it is helpful to define a cut as: an assignment of vertices into two non-empty sets and an edge is said to cross the cut if its two ends are in different sets.

Theorem: If we have a tree which is part of an MST and we add to it the min-weight edge across the cute separating the tree from the other vertices, then the result is still part of an MST.

Proof

Let f be the tree and let f' be an MST that f is part of (the condition of the theorem requires that such an f' exists). Let e be the minimum edge weight edge across the cut. We want to show that there is an MST that includes f U {e}.



If f' includes edge then we are done. If f' does not consider e. Let u and v be the vertices at either end of e and consider the path in f' between u and v (there must be a path since f' is a spanning tree). This path must cross the cut (since its ends are on different sides of the cut). Let e' be an edge in the path that crosses the cut. Now, let g be f' but with e added and e' removed.

It is clear that weight(g) ≤ weight(f') therefore g is a MINIMUM spanning tree if it is a spanning tree. Since all points on the other side of the cut must be connected to f', by removing e' and adding e it is clear that g is a spanning tree and all nodes remain connected (since there is a connection between u and v) and therefore g is a minimum spanning tree.

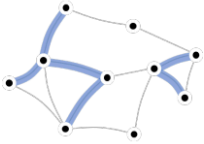
Kruskal's Algorithm

Problem Statement: Given a connected undirected graph with edge weights, construct an MST

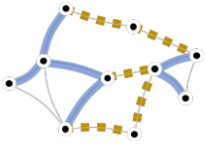
Kruskal's Algorithm makes the same assumptions as Prim's and solves the same problem. It has a worse running time, but produces intermediate states which can be useful.

Kruskal's algorithm builds up the MST by agglomerating smaller subtrees together. At each stage, we've built up some fragments of the MST (we start with each vertex being a fragment).

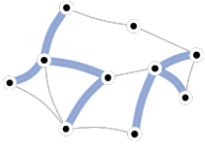
The algorithm greedily chooses two fragments to join together by picking the lowest-weight edge that will join two fragments.



four tree fragments have been found so far, including two trees that each consist of a single vertex

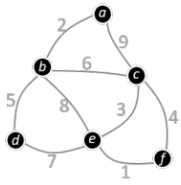


five candidate edges that would join two fragments

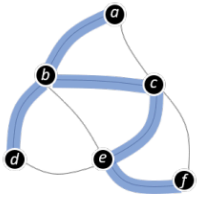


pick the cheapest of the five candidate edges, and add it, thereby joining two fragments

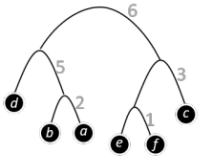
It is important to note that the operation of Kruskal's algorithm looks like clustering and intermediate stages correspond to a classification tree, which is where it's application is – for example in image segmentation.



an undirected graph with edge weights



the MST found by Kruskal's algorithm



draw each fragment as a subtree, and draw arcs when two fragments are joined

In the implementation of Kruskal's algorithm, we use a disjoint set, which keeps us keep track of a fragment – which vertices are in it.

```
def kruskal(g):
    tree_edges = []
    partition = DisjointSet()
    for v in g.vertices:
        partition.addsingleton(v)
    edges = sorted(g.edges, sortkey = lambda u,v,edgeweight: edgeweight)

    for (u,v,edgeweight) in edges:
        p = partition.getsetwith(u)
        q = partition.getsetwith(v)
        if p != q:
            tree_edges.append((u,v))
            partition.merge(p, q)
            # Let f be the forest made up of edges in tree_edges.
            # Assert: f is part of an MST
            # Assert: f has one connected component per set in partition

    return tree_edges
```

Running Time

We can say that all operations on a Disjoint Set can be done in O(1) time.

It is clear line 5 runs for every vertex and then it takes O(E log E) to sort the edges in line 6. Finally, there can me a maximum of V merges and we iterate over E edges. So the total cost is O(E + E log E + V) = O(E log E + V). Since the maximum number of edges in an undirected graph is V(V-1)/2 and the minimum number is V-1, we can say log E = Θ(log V) and so we can say it is O(E log V)

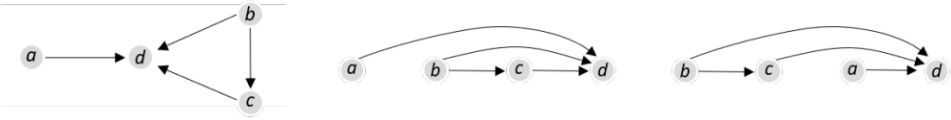
Correctness

To prove that Kruskal's algorithm finds an MST we apply the theorem used for the proof of Prim's algorithm as follows. When the algorithm merges fragments p and q, consider the cut of all vertices into p and not p, the algorithm picks a minimum weight edge cut across this cut and so by the theorem, we've still got an MST.

Topological Sort

Problem Statement: Given a directed acyclic graph (DAG), return a total ordering of all its vertices, such that if v1 → v2 then v1 appears before v2 in the total order

A directed graph can be used to represent ordering or preferences. We might then like to find a total ordering that's compatible.



The above graph has two orderings. It is clear that there is not a total order if there are cycles. Therefore, in a DAG, there will always have a total ordering.

The general idea is similar to a depth first search. When we reach a vertex, we visit all the children and other descendants, with v appearing before the descendants in the ordering.

```
1   def toposort(g):
2       for v in g.vertices:
3           v.visited = False
4           # v.colour = 'white'
5 +     totalorder = []  # an empty list
6       for v in g.vertices:
7           if not v.visited:
8               visit(v, totalorder)
9 +     return totalorder
10
11  def visit(v, totalorder):
12      v.visited = True
13      # v.colour = 'grey'
14      for w in v.neighbours:
15          if not w.visited:
16              visit(w, totalorder)
17 +    totalorder.prepend(v)
18      # v.colour = 'black'
```

Running Time

It is clearly still pretty much identical to depth first searching so the running time is clearly O(V + E).

Correctness

Theorem: The toposort algorithm terminates and returns totalorder which solves the problem statement.

Proof

Pick any edge v1 -> v2. We want to show that v1 appears before v2 in. total order. It's easy to see that every vertex is visited exactly once, and on that visit (1) it's coloured grey, (2) some stuff happens, (3) it's coloured black.

When v1 is coloured grey. At this instant, there are three possibilities for v2:

1. v2 is black. If this is so, then v2 has already been prepended on the list before v1, so v2 will appear after v1.
2. v2 is white. If this is so, then v2 hasn't been visited, therefore we'll call visit(v2) in line 16. This call must finish before returning to visiting v1, therefore v2 gets prepended earlier, therefore v1 will appear before v2 in total order
3. v2 is grey. If this is so, there was an earlier call to visit(v2) which we are inside. Therefore, there is a path between v2 and v1. However, since there is an edge from v1 to v2, this implies there is a cycle, which is impossible for a DAG, so we have reached a contradiction, so v2 cannot be grey.

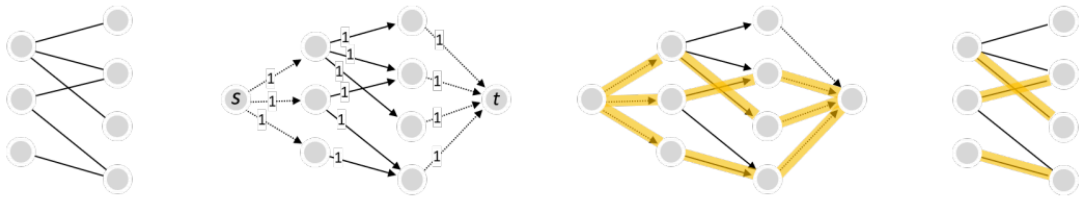
Networks and Flows

Matchings in bipartite graphs 18

A bipartite graph is an undirected graph where the vertices are set into two sets are all edges go between these sets.

A matching in a bipartite graph is a selection of edges such that no vertex is connected to more than one edge. The size of a matching is the number of edges it contains. A maximum matching is one with the largest possible size.

We find a maximum matching by turning it into a more complicated problem and solving that:



1. Firstly, we start with a bipartite graph
2. We add a source s with edges to each left-hand vertex; add a sink with edges from each right-hand vertex; turn the original edges into directed edges from left to right and give all the edges capacity one.
3. Run the Ford-Fulkerson algorithm to find a maximum flow from the source to the sink

4. Interpret that flow as a matching

Correctness

Theorem:

1. **The maximum matching algorithm above terminates**
2. **It produces a matching**
3. **There is no matching with larger size (i.e. it produces a maximum matching)**

Proof of (1): The proof of Ford-Fulkerson tells us that the Ford-Fulkerson algorithm terminates.
Proof of (2): Say f* is the flow produces by Ford-Fulkerson. The lemma tells us that f* is integer on all edges. Since the edge capacities are all 1, the flow must be 0 or 1 on all edges. Translate f* into a matching m* by selecting all the edges in the original bipartite graph that got f* flow of 1. The capacity constraints from the source means that each left hand vertex has either 0 or 1 going in so it must have 0 or 1 flow going out, therefore it is connected to at most one edge in m*. Similarly, each right-hand vertex is connected to at most one edge in m*. Therefore, m* is a matching.
Proof of (3): Consider any other matching m. We can translate m into a flow f. The translation between flows and matching means that: Size(m) = value(f) and size(m*) = value (f*) Since f* is a max flow, therefore value(f) ≤ value(f*) => size(m) ≤ size(m*) => m* is a maximum matching.

Max-flow min-cut theorem

Consider a directed graph with each edge having a label c(u -> v) > 0 called the capacity. Let there be a source vertex s and a sink vertex t. A flow is a set of edge labels f(u -> v) such that:

$$0 \leq f(u \to v) \leq c(u \to v) \; for \; each \; edge$$

and

$$\sum_{u:u\to v} f(u \to v) = \sum_{w:v\to w} f(v \to w) \;\; at \; all \; vertices \; v \in V$$

All this says is that all the flow going in goes out.

$$value(f) = \sum_{u:s\to u} f(s \to u) - \sum_{u:u\to s} f(u \to s)$$

A cut is a partition of the vertices into two sets V = S U S' with s ϵ S and t ϵ S'

The capacity of a cut is:

$$capacity\ (S, S') = \sum_{\substack{u\in S,\ v\in S': \\ u\to v}} c(u \to v)$$

Theorem (Max-flow min-cut theorem): For any flow f and any cut (S, S')

Value(f) ≤ capacity(S, S')

Proof

$$
\begin{aligned}
\text{value}(f) &= \sum_u f(s \to u) - \sum_u f(u \to s) && \text{by definition of flow value} \\
&= \sum_{v \in S} \left(\sum_u f(v \to u) - \sum_u f(u \to v) \right) && \text{by flow conservation} \\
&&& \text{(the term in brackets is zero for } v \neq s\text{)} \\
&= \sum_{v \in S} \sum_{u \in S} f(v \to u) + \sum_{v \in S} \sum_{u \notin S} f(v \to u) \\
&\quad - \sum_{v \in S} \sum_{u \in S} f(u \to v) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) \\
&&& \text{(splitting the sum over } u \text{ into two sums, } u \in S \text{ and } u \notin S \text{)} \\
&= \sum_{v \in S} \sum_{u \notin S} f(v \to u) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) && \text{by 'telescoping' the sum} \\
&\leq \sum_{v \in S} \sum_{u \notin S} f(v \to u) && \text{since } f \geq 0 \qquad (1) \\
&\leq \sum_{v \in S} \sum_{u \notin S} c(v \to u) && \text{since } f \leq c \qquad (2) \\
&= \text{capacity}(S, \bar{S}) && \text{by definition of cut capacity.}
\end{aligned}
$$

Ford-Fulkerson

Problem Statement: Given a weighted directed graph g with a source s and a sink t, find a flow from s to t with maximum value (also called a maximum flow).

Ford-Fulkerson works in two main steps:

1. Find an augmenting path
2. Augment the path by putting as much flow through the path as possible.

```
1  def ford_fulkerson(g, s, t):
2      # let f be a flow, initially empty
3      for u→v in g.edges:
4          f(u→v) = 0
5      # Repeatedly find an augmenting path and add flow to it
6      while True:
7          S = Set([s])  # the set of vertices to which we can increase flow
8          while there are vertices v∈S, w∉S with f(v→w)<c(v→w) or f(w→v)>0:
9              S.add(w)
10         if t in S:
11             pick any path p from s to t made up of pairs (v,w) from line 7
12             write p as s = v_0, v_1, v_2, ..., v_k = t
13             δ = ∞ # amount by which we'll augment the flow
14             for each edge (v_i,v_{i+1}) along p:
15                 if v_i→v_{i+1} is an edge of g:
16                     δ = min(c(v_i→v_{i+1}) − f(v_i→v_{i+1}), δ)
17                 else v_i←v_{i+1} must be an edge of g:
18                     δ = min(f(v_{i+1}→v_i), δ)
19             # assert: δ > 0
20             for each edge (v_i,v_{i+1}) along p:
21                 if v_i→v_{i+1} is an edge of g:
```

```
22                    f(v_i → v_{i+1}) = f(v_i→v_{i+1}) + δ
23                else v_i←v_{i+1} must be an edge of g:
24                    f(v_{i+1}→v_i) = f(v_{i+1}→v_i) − δ
25            # assert: f is still a flow (according to defn. in Section 6.2)
26        else:
27            break  # finished — can't add any more flow
```

The pseudocode does not tell us how to choose the path in line 11. One sensible idea is to pick the shortest path and this version is called the Edmonds-Karp algorithm. Another idea is to pick the path that makes delta as large as possible, and this is also due to Edmunds and Karp.

Termination

Theorem: If all capacities are integers, then the algorithm terminates and the resulting flow on each edge is an integer.

Proof: Initially, the flow on each edge is 0. At each execution of lines 13-18, we start with integer capacities and integer flow sizes, so we obtain δ to be an integer ≥ 0. Therefore, the total flow has increased by an integer after lines 20-24. The value of the flow can never exceed the sum of all capacities, so the algorithm must terminate with the flow always being an integer.

Running Time

We execute the while loop at most f' times, where f' is the value of the maximum flow. We can build the set S and find a path (using bfs) in O(V+E). Lines 13-24 involve operations per edge of the path which is O(V). Therefore the total running time is O(f'(E + V + V)) = O(f'(E+V)) Since E ≥ V – 1, we can write this as O(f'E). It is unsatisfactory having the running time in terms of f', but there is no way we can quantify this in terms of the edges and vertices.

Edmonds-Karp can be shown to have running time of $O(E^2V)$.

Correctness

Theorem: If the algorithm terminates, and f* is the final flow it produces, then:

1. The value of f* is equal to the capacity of the cut found in lines 7-9
2. f* is a maximum flow

Proof of 1: Let (S, S') be the cut. When the algorithm, terminates, by the condition on line 8, f*(w -> v) = 0 $\forall$ v ϵ S, w !ϵ S so inequality (1) of the previous page proof is an equality. By the same condition, f*(v -> w) = c(v -> w) so inequality (2) is always an equality. Therefore, value(f*) is equal to the capacity of the cut.

Proof of 2: The Max-Flow Min-Cut theorem states:

$$value(flow) \leq capacity(S, S')$$

Since by part 1, we have a flow with value equal to the capacity. Therefore, f* is a maximum flow.

A cut corresponding to a maximum flow is called a bottleneck cut. It may not be unique, but the value and capacity are unique.

Advanced Data Structures

Priority Queue ADT

In a priority queue, we keep track of a dynamic set of clients, each keyed with its priority and have the highest-priority one always be promoted to the front of the queue regardless of when it joined.

The structure must support the promotion of an item to a more favourable position in the queue.

1. Void insert(Item x)
2. Item first()
3. Item extractMin()
4. Void decreaseKey(Item x, Key new)
5. Void delete(Item x)

```
0  ADT PriorityQueue {
1    void insert(Item x);
2    // BEHAVIOUR: add item <x> to the queue.
3
4    Item first(); // equivalent to min()
5    // BEHAVIOUR: return the item with the smallest key (without
6    // removing it from the queue).
7
8    Item extractMin(); // equivalent to delete(), with restriction
9    // BEHAVIOUR: return the item with the smallest key and remove it
10   // from the queue.
11
12   void decreaseKey(Item x, Key new);
13   // PRECONDITION: new < x.key
14   // PRECONDITION: Item <x> is already in the queue.
15   // POSTCONDITION: x.key == new
16   // BEHAVIOUR: change the key of the designated item to the designated
17   // value, thereby increasing the item's priority (while of course
18   // preserving the invariants of the data structure).
19
20   void delete(Item x);
21   // PRECONDITION: item <x> is already in the queue.
22   // BEHAVIOUR: remove item <x> from the queue.
23   // IMPLEMENTATION: make <x> the new minimum by calling decreaseKey with
24   // a value (conceptually: minus infinity) smaller than any in the queue;
25   // then extract the minimum and discard it.
26 }
27
28 ADT Item {
29   // A total order is defined on the keys.
30   Key k;
31   Payload p;
32 }
```

A very basic implementation of the ADT would be using a sorted array, which would have the following performances:

Operation	Cost with sorted array
creation of empty queue	$O(1)$
`first()`	$O(1)$
`insert()`	$O(n)$
`extractMin()`	$O(n)$
`decreaseKey()`	$O(n)$
`delete()`	$O(n)$

Binary Heap

What a heap is, was covered in the description of heapsort. Here we use a min-heap rather than a max-heap. A min-heap is a binary tree which satisfies two additional invariants. It is "almost full" and the lowest level is filled left to right, and it obeys the heap property, that is each node has a key less than or equal to those of its children.

- As a consequence of heap property, the root is the smallest value – therefore to read out the highest priority item, we just look at the root (O(1)).
- To insert an item, we add it to the end of the heap and let it bubble up – O(H).
- To extract the root, we read it out, then replace it with last item and let that sink down – O(H)
- To reposition an item after decreasing the key, we let it bubble up towards the root – O(H)

Since the tree is balanced – it is always full, the height is always O(log n) therefore the asymptotic complexities are:

Operation	Cost with binary min-heap
creation of empty heap	$O(1)$
`first()`	$O(1)$
`insert()`	$O(\lg n)$
`extractMin()`	$O(\lg n)$
`decreaseKey()`	$O(\lg n)$
`delete()`	$O(\lg n)$

Amortized Analysis

For some data structures, looking at the worst-case run time per operation may be unduly pessimistic, especially in two situations:

1. Some data structures are designed so that most operations are cheap, but some of them have occasional expensive internal housekeeping.
2. Sometimes we want to know the aggregate cost of a sequence of operations, not the individual costs of each operation.

Aggregate Analysis: Working out the worst-case total cost of a sequence of operations.

Example: Consider a dynamically-sized array with initial capacity 1 which doubles when it becomes full.

Assume that the cost of doubling capacity from m to 2m is m. After adding n elements, the total cost from all the doubling is:

$$\mathbf{1} + \mathbf{2} + \cdots + \mathbf{2}^{\lfloor lg\, n-1 \rfloor}$$

which is ≤ 2n – 3. The cost of writing in the n values is n. Therefore, the total cost is ≤ 3n-3 which is O(n)

Amortized Cost

When we look at the aggregate cost of a sequence of operations, we can reassign costs like this:

$$c_1 + \cdots + c_j \leq c'_1 + \cdots + c'_j$$

Where c's are your amortized costs.

Example: Consider a dynamically-sized array with initial capacity 1 which doubles when it becomes full.

If we assign a cost c' = 3 to each append operation, we ensure that the true cost is less than the amortized cost we have said. (We said that n calls to append cost ≤ 3n – 3). Therefore, the amortized cost is O(1).

The Potential Method

We define a function ϕ, called a potential function, that maps possible states of the data structure to real numbers ≥ 0. For an operation with true cost c, for which the state just beforehand was S_{ante} and the state just after is S_{post}, define the amortized cost of the operation to be:

$$c' = c + \phi(S_{post}) - \phi(S_{ante})$$

Proof that c' is a valid amortized cost:

Consider a sequence of operations

$$S_0 - c_1 \to S_1 - c_2 \to S_2 - c_3 \to \cdots - c_k \to S_k$$

Aggregate Amortized Cost

$$\begin{aligned} &= \left(-\Phi(\mathcal{S}_0) + c_1 + \Phi(\mathcal{S}_1)\right) + \left(-\Phi(\mathcal{S}_1) + c_2 + \Phi(\mathcal{S}_2)\right) \\ &\quad + \cdots + \left(-\Phi(\mathcal{S}_{k-1}) + c_k + \Phi(\mathcal{S}_k)\right) \\ &= c_1 + \cdots + c_k - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k) \\ &= \text{aggregate true cost} - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k). \end{aligned}$$

Therefore we have proved this.

Example: Consider a dynamically-sized array with initial capacity 1 which doubles when it becomes full.

Define the potential function as:

$$\phi = [2(numOfItemsInArray) - capacityOfArray]^+$$

This potential function is ≥ 0, and for an empty data structure it is = 0. Now, consider the two ways that append could play out:

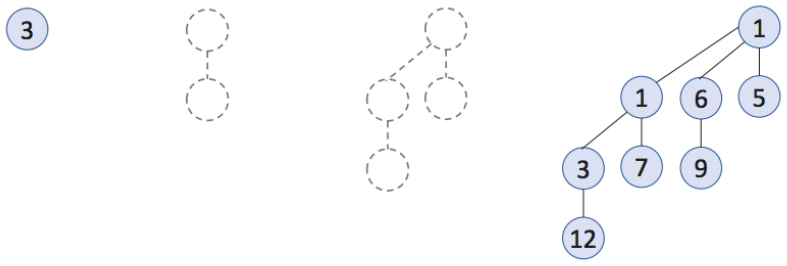
1. We could add the new item without needing to double the capacity. The true cost is c = O(1) and the change in potential is = 2, therefore the amortized cost = O(1) + 2
2. Alternatively, we need to double the capacity. Let n be the number of items just before. The true cost is O(n), to create the new array and copy n items and then write in the new value. The change in potential is 2 – n, so the amortized cost = O(n) + 2 – n = O(1)

In both cases, the amortized cost of an append is O(1)

Binomial Heap

A binomial heap is a compromise between the linked list and binary heap methods of keeping a priority queue. It maintains a list rather than obsessively tidying everything into a single heap so that push is fast, but it still keeps some heap structure so that popmin is fast. It is defined hence:

1. A binomial tree of order 0 is a single node. A binomial tree of order of order k is a tree obtained by combining two binomial trees of order k-1, by appending one of the trees to the root of the other
2. A binomial heap is a collection of binomial tree, at most one for each tree order, each obeying the heap property.
3. This is a binomial heap consisting of one binomial tree of order 0 and one of order 3. (The dotted parts simply say there are no trees of order 1 or 2).



Basic Properties

1. Binomial tree of order k has 2^k nodes and height k.
2. In a binomial tree of order k, the root node has k children – the degree of the root is k
3. In a binomial tree of order k, the root node's j children are binomial trees of order k-1, k-2, …, 0
4. In a binomial heap with n nodes, the 1s in the binary expansion of the number n correspond to the orders of trees contained in the heap.
5. If a binomial heap contains n nodes, it contains O(lg n) binomial trees and the largest of those trees has O(lg n)

Operations

1. Merge (h1, h2) – O(log n)
    a. To merge two binomial heaps, we start from order 0 and go up, as if doing binary addition, but instead of adding digits in place k, we merge binomial trees of order k, keeping the tree with smaller root on top. If n is the total number

of nodes in both heaps together, then there are O(log n) trees in each heap and O(log n) operations in total.

2. Push (v, k) – O(log n)
    a. Though it has true worst case cost of O(log n), it has an amortized cost of O(1)
        i. A series of adding a single order 0 tree is effectively looking for the first 0 in the sequence of the original heap each time. Equivalent to incrementing a binary counter and seeing which things need to be changed.
        ii. N changes to last, n/2 to second last, n/4 to next, etc
        iii. SumOfNumberOfChanges = GP with r = ½
        iv. $S = 2n(1 – 2(0.5)^k)$ {for a k order heap}
        v. Therefore amortized cost of O(1)
    b. Treat the new item as a binomial heap with only one node and merge it.
3. DecreaseKey (v, newK) – O(log n)
    a. Proceed, as with a normal binary heap, applied to the tree with v belongs, change and bubble up as needed, therefore O(log n)
4. PopMin() – O(log n)
    a. First, scan the roots of all the trees in the heap, at cost O(log n) to find which root to remove. Cut it out from its. It's children for a binomial heap by property 3 and merge this with the rest of the original heap at cost O(log n)

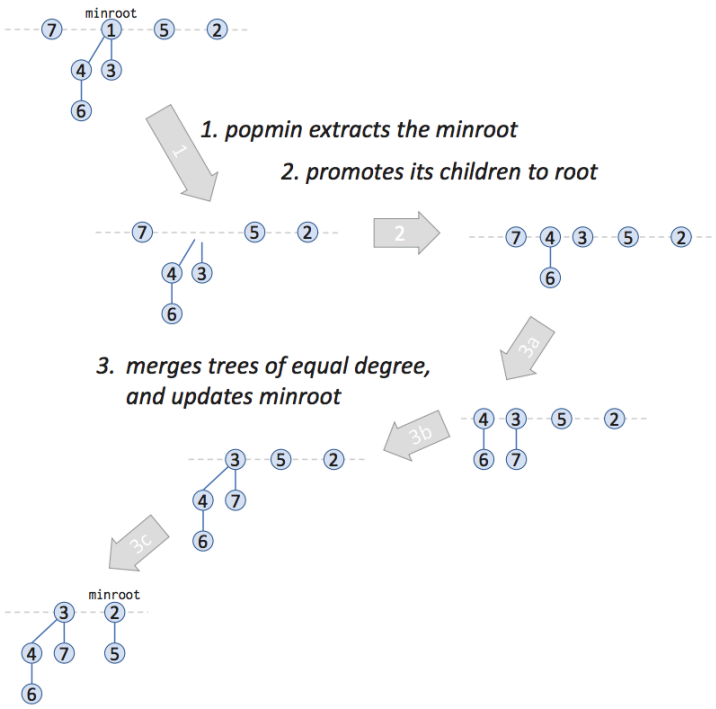
Fibonacci Heap

A fibonnaci heap is a fast priority queue, specifically designed to speed up Dijkstra's algorithm. The general idea is that we should just be lazy while we are doing push and decreasekey, only doing O(1) work, and just accumulating a collection of unsorted items, only doing cleanup on calls to popmin.

The general reasoning for this is that adding n items to a binary heap one by one is O(n log n), but only O(n) to heapify them together.

The other big idea is for decreasekey to be O(1), by only touching a small part of the datastructure.

Push and Popmin




```
1  # Maintain a list of heaps (i.e. store a pointer to the root of each heap)
2  roots = []
3
4  # Maintain a pointer to the smallest root
5  minroot = None
6
7  def push(v, k):
8      create a new heap h consisting of a single item (v, k)
9      add h to the list of roots
10     update minroot if k < minroot.key
11
12 def popmin():
13     take note of minroot.value and minroot.key
14     delete the minroot node, and promote its children to be roots
15     # cleanup the roots
16     while there are two roots with the same degree:
17         merge those two roots, by making the larger root a child of the smaller
18     update minroot to point to the smallest root
19     return the value and key from line 13
```

Cleanup

```
20 def cleanup(roots):
21     root_array = [None, None, ...]  # empty array
22     for each tree t in roots:
23         x = t
24         while root_array[x.degree] is not None:
25             u = root_array[x.degree]
26             root_array[x.degree] = None
27             x = merge(x, u)
28         root_array[x.degree] = x
29     return list of non-None values in root_array
```

Decrease Key

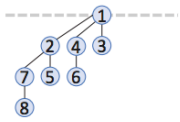
If we can decrease the key of an item in-place (parent is still less than new key), then that's all that decrease key needs to do. If however, the node's new key is smaller than the parent, we need to do something to maintain the heap. However, if we just cut out nodes and dump them in the root list, we might end up with trees that are shallow and wide even as big as O(n), making popmin very costly.

Therefore, we need to keep the maximum degree small by two rules:

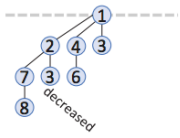
1. Lose one child, and you become a 'marked' node (sometimes called a 'loser' flag)
2. Lost two children, you get moved into the root list (and your mark is removed.

```
30 # Every node will store a flag, p.loser = True / False
31
32 def decreasekey(v, k'):
33     let n be the node where this value is stored
34     n.key = k'
35     if n violates the heap condition:
36         repeat:
37             p = n.parent
38             remove n from p.children
39             insert n into the list of roots, updating minroot if necessary
40             n.loser = False
41             n = p
42         until p.loser == False
43         if p is not a root:
44             p.loser = True
45
46 def popmin():
47     mark all of minroot's children as loser = False
48     then do the same as in the simple version, lines 13–19
```

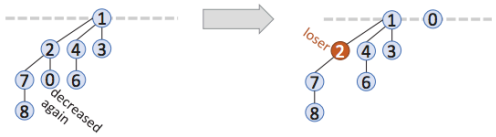
Example:



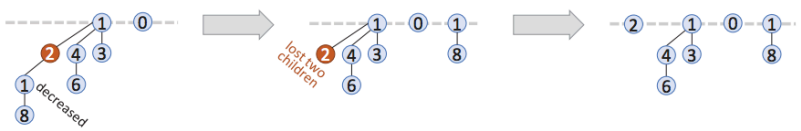
decreasekey from 5 to 3



decreasekey again to 0 — move 0 to maintain the heap

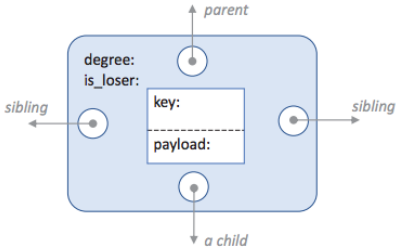


decreasekey from 7 to 1 — move 1 to maintain the heap — move the double-loser to root



Implementing a Fibonacci Heap

We generally use a circular doubly-linked list for the root list; and the same for a sibling list; and we'll let each node point to its parent and each parent will point to one of its children:



Analysis of Fibonacci Heap

Define the potential function as:

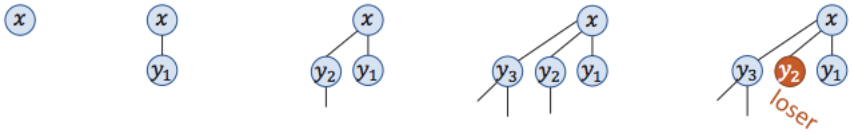
ϕ = number of roots + 2(number of loser nodes)

Let n be the number of items in the heap, and let d_{max} be an upper bound on the degree of any node in the heap.

1. **Push:** amortized cost O(1)
    a. This just adds a new node to the root list so the true cost is O(1). The change in potential is 1 so the amortized cost is O(1)
2. **Popmin:** amortized cost $O(d_{max})$
    a. First, cut out minroot and promote its children to the rootlist. There are at most d_{max} to promote, so the true cost is $O(d_{max})$. These children get promoted to root and maybe some of them lose the loser mark so $\Delta\phi \leq d_{max}$. So the amortized cost for this first part is $O(d_{max})$.
    b. The second part is running cleanup. Line 21 initialises an array and size $d_{max} + 1$ will do, so this $O(d_{max})$. Suppose the cleanup does t merges and ends up with d trees; there must have been d+t trees to start with; the total amortized cost is O(d), and $d \leq d_{max}$
    c. Therefore, total process is $O(d_{max})$
3. **DecreaseKey:** amortized cost O(1)
    a. O(1) elementary operations to decrease the key. If the node doesn't have to move, then $\Delta\phi = 0$ and so amortized cost = O(1)
    b. If the node does have to move, the following happens
        i. We move the node to the root list – the true cost is O(1) and ϕ increases by ≤ 1 (increases by 1 if the node wasn't a loser and decreases by 1 if it was)
        ii. Some of the root's loser ancestors (say B and C) have to be moved to the root list and ϕ is increased by 1 and then decreased by 2. Therefore the amortized cost of this part is zero, regardless of the number of loser ancestor.
        iii. One ancestor might have to marked as a loser – the true cost is O(1) and the $\Delta\phi = 2$. Therefore amortized cost = O(1)

Theorem: If a node in a Fibonacci heap has d children, the the subtree rooted at that node consists of $\geq 1.618^d$ nodes. Precisely, it has $\geq F_{d+2}$ nodes, the (d+2)nd Fibonacci number. As a corollary of this, we can clearly say that $d_{max} = O(\lg n)$

Proof: Consider an arbitrary node x in a Fibonacci heap, at some point in execution and suppose it has d children, call them y1, y2, …, yd in the order of when they last became children of x.



When x acquired y2 as a child, x already had y1 as a child, so y2 must have had ≥ 1 child seeing as it got merged into x. Similarly, when x acquired y3, y3 must have had ≥ 2 children, and so on. After x acquired a child yi, that child might have lost a child but it can't have lost more because of the rules of decrease key. Therefore, at the point of execution at which we're inspecting x:

y1 has ≥ 0 children
y2 has ≥ 0 children
y3 has ≥ 1 child
yd has ≥ d-2 children

Consider an arbitrary tree all of whose nodes obey the grandchild rule "a node with children I = 1, …, d has at least i-2 grandchildren via child i". Let N_d be the smallest possible nodes in a subtree whose root has d children. Then:

$$N_d = \underbrace{N_{d-2}}_{\text{child } d} + \underbrace{N_{d-3}}_{\text{child } d-1} + \cdots + \underbrace{N_0}_{\text{child 2}} + \underbrace{N_0}_{\text{child 1}} + \underbrace{1}_{\text{the root.}}$$

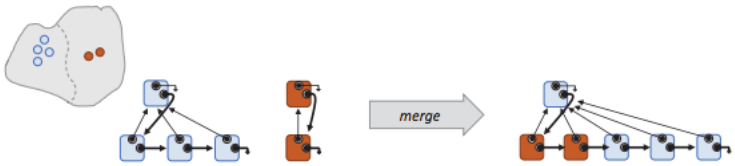
Substituting in N_{d-1}, we get $N_d = N_{d-2} + N_{d-1}$, the defining equation for the Fibonacci sequence, hence $N_d = F_{d+2}$.

Disjoint Sets

A Disjoint set is used to keep track of a dynamic collection of items in disjoint sets. We used it in Kruskal's algorithm.

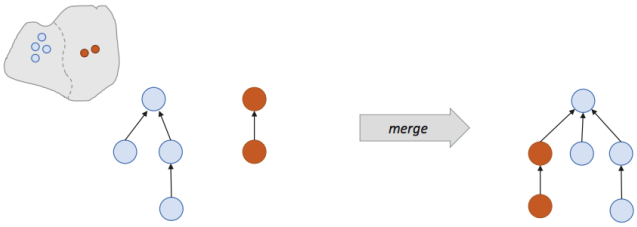
```
ADT DisjointSet:
    # Holds a dynamic collection of disjoint sets

    # Return a handle to the set containing an item.
    # The handle must be stable, as long as the DisjointSet is not modified.
    Handle get_set_with(Item x)

    # Add a new set consisting of a single item (assuming it's not been added already)
    add_singleton(Item x)

    # Merge two sets into one
    merge(Handle x, Handle y)
```

Flat Forest



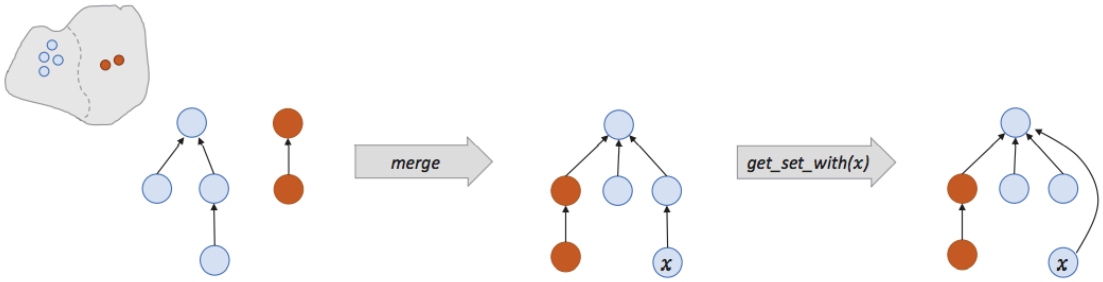
- In order to make get_set_with fast, we could make each item to point to its set's handle.
- This makes get_set_with a single lookup – O(1).
- However, merge needs to iterate through each item in one or other set and update its pointer and therefore takes O(n) time.
- We can keep track of the size of each set and pick the smaller set to update.
    - **Weighted union heuristic.**

Deep Forest



- To make merge faster, we could skip all the work of updating the items in a set and just build a deeper tree.
- Here, merge attaches one root to the other, which only requires updating a single pointer O(1).
- However, get_set_with needs to walk up the tree to find the root. This takes O(h) time, where h is the height of the tree.
- To keep h small, we can use the same idea as for the flat forest.
    - We can keep track of the height of the tree for each root (its rank) and always attach the lower-rank root to the higher ranking.
    - If the two roots had rank r1 and r2, then the resulting rank would be max(r1, r2) if r1 != r2 and r1 + 1 if r1 = r2
    - **Union by rank heuristic**

Lazy Forest



- This defers clean-up until you need it to get the answer
- Merge works as with for the deep forest, therefore being O(1)

- Get_set_with does some clean-up. It walks up the tree once to find the root, and then walks up the tree a second time and makes all nodes found to be intermediate nodes to be direct children of the root.
    - This is called the **path compression heuristic**
    - Won't adjust the stored ranks during path compression, so rank wouldn't be the exact height, just an upper bound.
- The cost of m operations on n items is O(mα_n) where α_n is an integer-valued monotonically increasing sequence, related to the Ackerman function, which grows extremely slowly

$$\alpha_n = 0 \quad \text{for } n = 0, 1, 2$$
$$\alpha_n = 1 \quad \text{for } n = 3$$
$$\alpha_n = 2 \quad \text{for } n = 4, 5, 6, 7$$
$$\alpha_n = 3 \quad \text{for } 8 \leq n \leq 2047$$
$$\alpha_n = 4 \quad \text{for } 2048 \leq n \leq 10^{80}, \text{ more than there are atoms in the observable universe.}$$

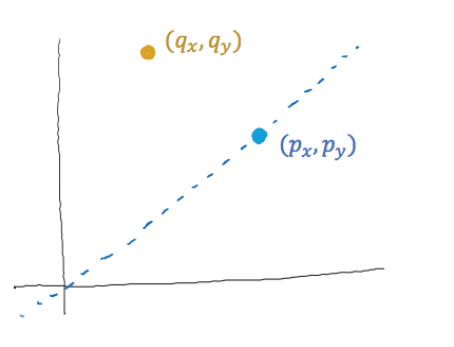
    - For practical purposes, α_n can be ignored in O notation and therefore the amortized cost per operation is O(1)

Geometric Algorithms

Segment Intersection

Testing if a point is above or below a line:

Example: test if q is above the dotted line



Possibility 1: if $q_y > (p_y/p_x)q_x$ then it's above

Possibility 2: Let $p^T = (-p_y, p_x)$. Now the sign of $p^T \cdot q$ tells us which side of the dotted line q is on – easy to show based on it considering the angle between p^T and q

Testing if two line segments ($\overrightarrow{rs}$ and $\overrightarrow{tu}$) intersect:

1. If t and u are both on the same side of the extension of rs ie if $(s-r)^T \cdot (t-r)$ and $(s-r)^T \cdot (u-r)$ have the same sign, then the two line segments don't intersect.
2. Otherwise, if r and s are on the same side of the extension of tu, then the two line segments don't intersect
3. Otherwise, they do intersect

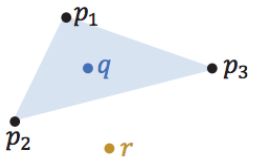
OR, just attempt to find a solution for the point they meet and see if there is a solution.

Jarvis' March

A convex combination is any vector:

$$q = \alpha_1 p_1 + \cdots + \alpha_n p_n \quad \text{where } \alpha_i \geq 0 \text{ for all } i, \text{ and } \sum_{i=1}^{n} \alpha_i = 1.$$

The convex hull of a collection of points is the set of all convex combinations



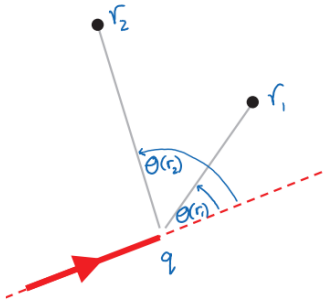
$q = 0.6p_1 + 0.16p_2 + 0.24p_3$: is a convex combination
$r = -0.5p_1 + 0.9p_2 + 0.6p_3$: not a convex combination

Jarvis' march allows us to compute the convex hull of a collection of points. (It actually finds the corner points of the convex hull which could be joined together to form the hull itself).

```
let q_0 be the point with lowest y–coordinate
(in case of a tie, pick the one with the largest x–coordinate)

draw a horizontal (left→right) line through q_0
for all other points r ∈ P:
    find the angle θ(r) from the horizontal line to q_0 r, measured ↺
let q_1 be the point with the smallest angle
(in case of a tie, pick the one furthest from q_0)

h = [q_0, q_1]
repeatedly:
    let p and q be the last two points added to h respectively
    for all other points r ∈ P:
        find the angle θ(r) from the extended p q line to q r, measured ↺
    pick the point with the smallest angle, and append it to h
    (in case of a tie, pick the one furthest from q)
    stop when we return to q_0
```

At every step of the iteration, we search for the point with the smallest angle, such that we cover the entire space. It is clear that the algorithm is O(nH) where n is the number of points and H is the number of points in the convex hull.

N.B. You don't necessarily need to find the angle, you can instead do this by checking which point is not on the left of any extended line from q, while all other points are on the left of it, hence:



This saves time in calculating thetas themselves

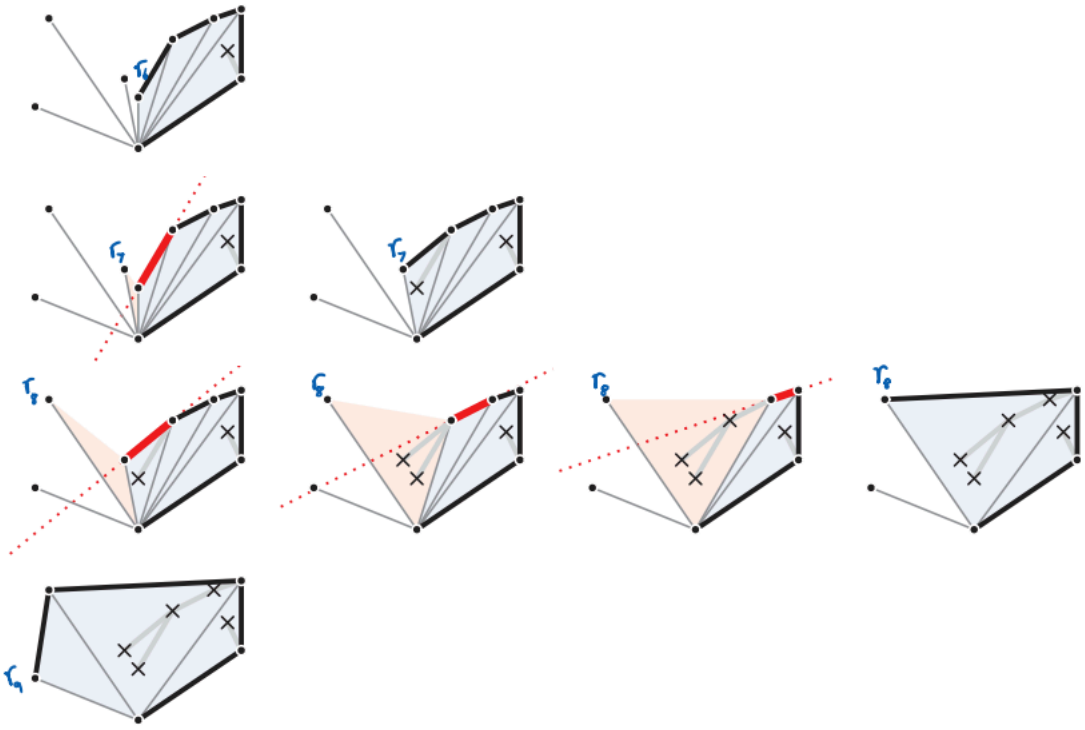
It is important to note that Jarvis' march is very much like selection sort: repeatedly finding the next item that goes into the next slot.

Graham's Scan

This algorithm also computes the convex hull, building it up by scanning through all the points, backtracking when necessary.

```
let r_0 be the point with lowest y–coordinate
(in case of a tie, pick the one with the largest x–coordinate)

draw a horizontal (left→right) line through r_0
for all other points r:
    find the angle from the horizontal line to r_0 r, measured ↺
let r_1,...,r_{n-1} be the sorted list of points, lowest angle to highest

h = [r_0, r_1]
for each r_i in the sorted list of points, i ≥ 2:
    if r_i isn't on the left of the extension of the final segment of h:
        # backtrack
        repeatedly delete points from the end of h until r_i is
    append r_i to h
```

This part of an example helps to make it obvious what is going on:



Analysis

Finding the angle for all points is O(n), while sorting them is O(n log n). During the scan, each point is added to the list once and can be removed once, so the loop is O(n). Therefore, in total, it is O(n log n).

Exam Suggestions