

# Part IB Revision Notes

## Data Structures And Algorithms

Michael Smith

April 22, 2004

## 1 Costs of Algorithms

- Asymptotic bounds on algorithms:
  - Upper bound:  $O(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n \geq n_0. 0 \leq f(n) \leq cg(n)\}$ .
  - Lower bound:  $\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n \geq n_0. 0 \leq cg(n) \leq f(n)\}$ .
  - Tight bound:  $\Theta(g(n)) = \{f(n) \mid \exists c, c', n_0 > 0. \forall n \geq n_0. 0 \leq cg(n) \leq f(n) \leq c'g(n)\}$ .
- For a divide and conquer approach to sorting, we have a cost  $f(n) = kn + 2.f(\frac{n}{2}) = O(n \log n)$ .
- In general, for recurrences of the form  $T(n) = aT(\frac{n}{b}) + f(n)$ , we can use the master theorem. Informally, if  $f(n)$  grows slower than the number of leaves in the recursion tree, then the dominant term is given by all the little bits of work done at the leaves –  $O(n^{\log_b a})$  (there are  $n^{\log_b a}$  leaves). If  $f(n)$  grows faster, then the dominant term is given by the large pieces of work done earlier in the recursion –  $O(f(n))$ . If they grow at the same rate, then we must consider the work done in earlier nodes (hence the  $\log_2 n$  factor). Formally (where  $\epsilon > 0$ ):
  - If  $f(n) = O(n^{\log_b a - \epsilon})$  then  $T(n) = \Theta(n^{\log_b a})$ .
  - If  $f(n) = \Theta(n^{\log_b a})$  then  $T(n) = \Theta(n^{\log_b a} \log_2 n)$ .
  - If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  then  $T(n) = \Theta(f(n))$  (so long as  $af(\frac{n}{b}) = O(f(n))$ ).

## 2 List and Table Data Structures

- The **List** datatype should support the following operations:
  - `make_empty_list()` – constructor.
  - `is_empty_list(s)` – predicate returns `true` if `s` is an empty list.
  - `cons(x, s)` – adds the element `x` to the front of the list `s`.
  - `first(s)` – returns the first element of the list `s` (`car` in Lisp).
  - `rest(s)` – returns the list `s` excluding the first element (`cdr` in Lisp).
  - `set_rest(s, s')` – replaces the rest-field in list `s` with list `s'`.
  - `equal(s, s')` – ambiguous; could be pointer or element-content equality.

- A **double-linked list** has pointers from each element in both directions. In addition to the `rest()` function we have a `previous()` function. This requires suitable end markers, and for the pointers to be updated after each operation.
- A **stack** is a LIFO (last in, last out) data structure, with `push()` and `pop()` operations for inserting and removing elements from the data structure. A **queue** is similar, except that the item removed is that which has been in the data structure for the longest (FIFO).
- The **Table** datatype should support the following operations:
  - `clear_table()` – deletes the contents of the table.
  - `set(key, value)` – adds an item to the table.
  - `get(key)` – returns the value from the last `set()` operation for the given key in the table.
- A table can be implemented as:
  - A **vector** – the table is stored as a simple array, with the key being the array index. The `set()` and `get()` operations are thus  $O(1)$ , and clearing the table will be  $O(n)$ . This only works if all the keys are in an integer range such that the number of keys used is not much smaller than the range (otherwise it is inefficient in space usage).
  - A **linked list** – elements are (key, value) pairs. The `set()` and `clear()` operations are both  $O(1)$ , as we just add an element to the start of the list, or we set the start pointer to `Nil`. A lookup is  $O(n)$  as we do a linear search through the list.
  - A **sorted vector** – store (key, value) pairs in a sorted array (by key), with some marker to indicate the last element (current size). The `get()` uses a binary search, so is  $O(\log n)$ . The `set()` operation is  $O(n)$  as we need to shift, on average, half the elements in order to insert in the right place. Clearing the table is  $O(1)$  (reset the end-marker).
  - A **binary tree** – store the (key, value) pairs so that the left subtree contains keys less, and the right subtree greater than, than the current key. The `set()` and `get()` operations are then  $O(\log n)$  for a balanced tree (unbalanced, pathological case is  $O(n)$ ). Clearing is  $O(1)$ .
  - A **hash table** – there is a hash function  $h()$  mapping each key to a bucket in the range  $[0, B - 1]$ . For an ‘open’ hash table, each bucket is a linked list of (key, value) pairs, which is small enough for a fast search (`set()` and `get()` are  $O(1)$  so long as the number of items stored is much smaller than the size of the hash table). A ‘closed’ hash table works by allowing only one entry in each bucket. If there is a clash, there is a rule for finding a different location, which is followed until a free location is found.

### 3 Memory Management

- We need a mechanism for **allocating memory** from the heap (e.g. `malloc` and `free` in C, `new` in Java/C++). Space usage information is most commonly kept in some form of free-list (a linked list of free nodes that can be allocated):
  - **First fit** – choose the first free block that is at least the required size  $n$ . A variant is ‘next fit’, which starts the search from a running pointer.

- **Best fit** – choose the smallest free block that is at least the required size  $n$ . This tends to result in greater fragmentation than with ‘first fit’.
- **Buddy system** – initially treat the whole memory space as one block. If there is a request for less than half the size of the biggest block, divide it in two (repeating until we have the closest size). We keep a separate free-list for each block size. When memory is deallocated adjacent blocks that were split (‘buddies’) can be recombined if they are both free. Using Fibonacci block sizes leads to less memory wastage, but trickier recombination.
- The bigger challenge is in **deallocating memory**, as we need to know when it is safe to reclaim some memory (if we are not explicitly told):
  - We can use **reference counts**, to determine whether a block can be freed or not. Each block maintains a count of how many external references there are to it. When a block is deallocated, decrement the counts on blocks that it references, and deallocate those if their count becomes zero. This doesn’t take into account cyclic data structures.
  - **Mark and sweep** – when there is too little free storage remaining, trace through all references to blocks, and mark them as reachable. We then perform a linear scan through the store, deallocating any block that isn’t marked (e.g. by adding it to the free-list).
  - **Stop and copy** – similar to ‘mark and sweep’, but when we trace through all the live data, we copy each block to a new, ‘borrowed’, area of storage. The memory references in the copied blocks must all be updated. We then continue running the program in the new storage space, and reclaim all the old space.
- **Ephemeral garbage collection** works on the principle that most data either dies young or lives (almost) forever. Allocated blocks get put initially in a short-term pool, with frequent garbage collection. If it survives this, it gets moved to a longer-lived pool that is garbage collected less often. There can be a number of levels of pool.

## 4 External Media Storage

- For physical storage, the most dominant factor is the time taken for disk accesses. A data structure must therefore try to minimise the number of blocks read from disk (assuming the whole dataset is too big to fit in memory).
- A **B-tree** is a perfectly balanced tree whereby each node contains  $n$  keys, and pointers to  $n + 1$  child nodes. All the leaves are at the same depth in the tree. The keys are sorted in ascending order, and each key has some ‘satellite information’ associated with it (data, or a pointer):
  - To search a B-tree, perform an  $n + 1$  way branching decision at each node. Note that the keys in child  $m$  must be between keys  $k_{m-1}$  and  $k_m$  in the parent (if both exist).
  - To add a key, find the node that should contain its key. If there is room, put the key and data there, otherwise split the block into two (sharing the keys equally), adding the median key to the parent node (repeating recursively).

- **Dynamic hashing** is used to retrieve any data item from the store in just one disk access. We have two hash functions, the first of which hashes a key to a disk block, and the other gives a ‘signature’. There is a memory resident table listing the maximum signature for each disk block:
  - To add a record, compute its hash. If the signature is no greater than the maximum signature for the block, then write it to that block. If the block is full, however, we take the record(s) with the largest signature in the block, compute a second hash, and move them into the new block. The maximum signature value for the old block is set to one less than the signature of the moved records.
  - To retrieve a record, compute its hash. If the signature is no greater than the maximum signature for the block, fetch that block. Otherwise, compute a second hash of the key, and repeat until the correct block is found.

## 5 Sorting Algorithms

- If we know nothing of the data we are sorting, the inherent complexity of the problem is  $\Theta(n \log n)$ . We must have performed some number of comparisons leading to the correct permutation. So, given  $n!$  permutations, and assuming a binary decision on each comparison, there must be a path of at least  $\log_2(n!)$  to always lead to the correct result, which is  $\Theta(n \log n)$ .
- **Insertion sort** works by going through each element in turn, and inserting it into the correct position. On average, an element originally at position  $j$  will be moved  $\frac{j}{2}$  spaces, hence we require  $\frac{j}{2}$  exchanges. The average and worst cases are  $O(n^2)$  exchanges. The best case is  $O(1)$  exchanges, if already sorted.
- **Shell's sort** is a variant of insertion sort, which has been proven  $O(n^{\frac{3}{2}})$  for some sequences, but is almost certainly better:
  - Choose a decreasing sequence  $S$ . Sedgewick suggests  $\dots, 40, 13, 4, 1$  as this is good for selecting different sets of the list each time. The starting value  $s \in S$  is the first such that  $s < n$ .
  - Take a subset of the values at  $1, s + 1, 2s + 1, \dots$  and perform insertion sort on it. Repeat for each  $i, s + i, 2s + i, \dots$  ( $1 < i < s$ ).
  - Repeat the previous step for each subsequent  $s$  in the sequence. After finishing  $s = 1$  the list is sorted.
- **Quick sort** works by firstly choosing a pivot value in the array,  $m$ . The array is then partitioned such that to the left of  $m$  are values  $\leq m$ , and to the right are values  $\geq m$ . The algorithm is recursively applied to the two partitions:
  - The algorithm is implemented using two pointers - one moving left from the start of the array, and one moving right from the end of the array. Whenever the left pointer points to a value  $> m$ , and the right  $< m$ , swap the two values. Thus the partitioning operation is  $O(n)$ .
  - The cost of quick sort (where  $i$  is the partition point) is given by the recurrence  $C(n) = kn + C(i) + C(n - i)$ . Similarly, the average cost is given by the recurrence  $C(n) = kn + \frac{1}{n} \sum_{i=1}^n (C(i - 1) + C(n - i))$ . This average case is  $O(n \log n)$ , but the worst case is  $O(n^2)$  if each pivot is already the biggest/smallest value.

- One optimisation to avoid the worst case, is to take the pivot as the median of the first, last and middle values in the array. If lots of the elements are equal, then it makes more sense to partition into three regions ( $< m$ ,  $= m$  and  $> m$ ). This can be done with four pointers, keeping the  $= m$  sections at the start and end of the array, then moving to the middle when partitioning is complete.
- **Heap sort** uses a heap data structure, such that each element is numbered in a breadth first manner - for an element  $x$  at index  $k$ , its children  $y$  and  $z$  are at indices  $2k$  and  $2k+1$  respectively. Furthermore,  $x \geq y$  and  $x \geq z$ . This is represented as an array, such that  $k$  is the index into the array:
  - The basic `heapify(A, i)` operation puts the element in array  $A$  at index  $i$  in the correct position in the heap. Assuming that the subtrees below  $i$  are heaps, then the subtree with root at  $i$  is now a heap. If the element at  $i$  is less than either of the left and right elements, swap with the largest of the elements, and recurse on that subtree. This operation is  $O(\log n)$  where  $n$  is the size of the subtree to heapify.
  - The initial heapify operation converts an array of values into a heap. It operates by calling `heapify()` in decreasing index on the top half of the array (the bottom half are already one-element heaps). Note that for a node at depth  $k$ , `heapify()` will be called at most  $O(d - k)$  times. There are at most  $2^k$  nodes at this depth, the overall cost is  $O(2^d \sum_{k=1}^d \frac{k}{2^k})$  (where  $d = \log n$ ), which is  $O(n)$ .
  - The algorithm works by taking the top item in the heapified array (the largest value) and swaps it to the end of the array (its correct position). The remaining data is heapified, and the process repeats until the heap contains just one element. Each heapify is  $O(\log n)$ , so the overall cost of heap sort is  $O(n \log n)$ .
- **Merge sort** uses a divide and conquer approach. The array is divided in half, and this repeats recursively until each subarray contains only one element. Following this, the subarrays are merged (using some extra intermediate storage) in sorted order, until we get back to the original array (now sorted). The merge operation is  $O(n)$ , since the subarrays to be merged are already sorted, so the overall cost of merge sort is  $\Theta(n \log n)$ .
- We can sort in linear time if we know something about the data. For a large number of values over a small range, we can simply count how many times each value appears, then write each out the correct number of times in ascending order. For  $n$  uniformly distributed random numbers (0.0 to 1.0), we can calculate an index into an array of, say, size  $2n$ . Insert each item in the approximate place (or if already occupied the nearest place), then use insertion sort to clear up the mess.
- **Radix sort** is a fast method for sorting values in some base  $b$ , of digit-length  $l$ :
  - From the most significant end, place the values into  $b$  pigeon holes according to the most significant digit. Repeat within each pigeon hole for each digit (creating sub-pigeon holes). Finally, write out the elements in pigeon hole order. This results in the problem of lots of intermediate arrays to keep track of.
  - From the least significant end, place the values into pigeon holes according to their least significant digit. Append together these pigeon holes in order, to give one array. Repeat for each digit. Note that for the  $k$ th digit, each pigeon hole is sorted with respect to the least significant  $k - 1$  digits, so long as we pass through the array in index order. This is  $\Theta(ln)$ .

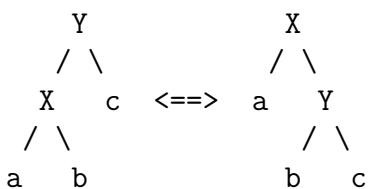
- **Order statistics** are used to find the median of a set of values without sorting them:
  - Variant of quicksort – partition into three sections ( $< m$ ,  $= m$ ,  $> m$ ). If the middle element in the array is in the ‘ $= m$ ’ section,  $m$  is the median. If not, recurse on the section that the middle element is in. This yields a recurrence  $C(n) \simeq kn + C(\frac{n}{2})$ , which is  $O(n)$  in the average case, but tends to  $O(n^2)$  in the worst case.
  - The method of **quintuplets** is  $O(n)$  in the worst case. Split the array into groups of five elements, and sort each (this step is  $O(n)$ ). Find the median of these medians (by using this algorithm recursively). Now, use the quicksort method, using this value as the pivot (at each stage finding a new pseudo-median as pivot). This guarantees that in the worst case we reduce the problem to  $\frac{7}{10}$  of the original, and therefore it is  $O(n)$  in the worst case (the pivot has  $\frac{n}{10}$  medians smaller than it as there are  $\frac{n}{5}$  medians in total. Each median has two elements smaller than it, so the ‘ $< m$ ’ and ‘ $> m$ ’ sections cannot be smaller than  $\frac{3n}{10}$ ).

## 6 Sets and Trees

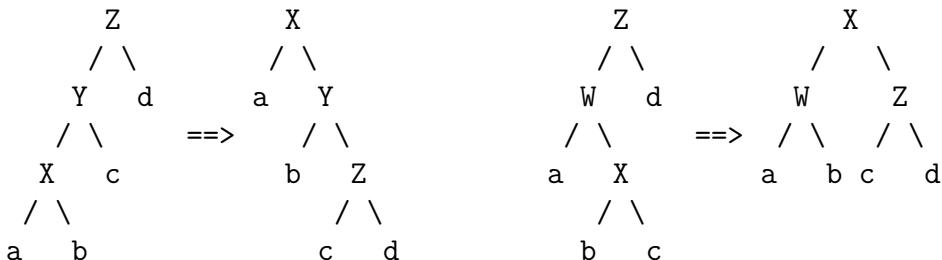
- The **Set** datatype should support the following operations:
  - `make_empty_set()` – constructor.
  - `is_empty_set(s)` – predicate returns `true` if `s` is an empty set.
  - `choose_any(s)` – returns an arbitrary item from `s`.
  - `insert(x,s)` – adds the item `x` to the set `s`(modifying `s`).
  - `search(s,k)` – returns the item with key `k` if it is in `s`.
  - `delete(s,k)` – remove the item with key `k` from `s` (if already in the set).
  - `minimum(s)` – returns the item in `s` with the smallest key [ordered sets].
  - `maximum(s)` – returns the item in `s` with the largest key [ordered sets].
  - `successor(s,x)` – returns the item with next larger key than `x` [ordered sets].
  - `predecessor(s,x)` – returns the item with next smaller key than `x` [ordered sets].
  - `union(s,s')` – return a new set containing all elements of `s` and `s'`.
  - `intersection(s,s')` – return a new set of only elements present in both `s` and `s'`.
- **2-3-4 trees** are a special case of B-trees, such that we have 2-nodes (1 key and 2 pointers), 3-nodes (2 keys and 3 pointers) and 4-nodes (3 keys and 4 pointers). All the keys  $k$  from a pointer between keys  $k_1$  and  $k_2$  are such that  $k_1 \leq k < k_2$ :
  - To add an item, find the lowest-level node that it should be placed in. During the search, if any 4-nodes are encountered, split them into two 2-nodes, adding the middle key to the parent node. When the correct node is reached (and split if necessary), add the key to it.
  - The tree remains balanced, as the only time that the length of a path from the root to a leaf changes is when the root node is split (all paths increase by 1). Thus the `search()` operation is  $O(\log n)$  in the worst case.
- **Red-black trees** are an alternative implementation of 2-3-4 trees, such that we only have 2-nodes, but each node has a colour. All the leaves (`nil`) are black, and a red node must always have black children. There must be the same number of black nodes on any root-to-leaf path (i.e. the ‘black-height’ of the tree is constant):

- To convert from a 2-3-4 tree, a 2-node is a normal black node, a 3-node is a black node with one red child, and a 4-node is a black node with two red children. Red nodes can be thought of as ‘virtual’ pointers, in that they extend a black node.
- To split a 4-node (black with two red children) just invert the colours of the three nodes. If this results in two consecutive red nodes, firstly get the two red nodes in the same direction (e.g. left-left), using the appropriate rotate operation [see splay trees], then perform a rotate on the parent node and change the colours so that the new parent is black and its children are red.
- To insert a node, add it as a red node in the correct place, then perform split/rotate operations to restore the red-black tree invariants.
- **Priority queues** are most often implemented using a heap, such that the minimum element can be extracted easily. Binomial and fibonacci heaps provide efficient support for the `union()` operation.
- **Ternary trees** are most commonly used for containing dictionaries. Each node has a left branch, a right branch, and a successor. If each node contains a character, then the successor gives the set of characters (from its left/right pointers) that may follow that character. It therefore stores words efficiently, and is good for doing crossword lookups (e.g. M?NK?Y).
- **Splay trees** are self-updating binary trees, such that all operations (insert, update, lookup, delete) have an amortized cost of  $O(\log n)$ . It only differs in efficiency by a constant factor from any specifically designed tree datastructure. When an update, delete or lookup operation is performed, we bring the node to the top of the tree using the transformations below. For an insert, we move the smallest item greater than or equal to the one we are inserting to the root, so that the left subtree from the root is less than the item (which becomes the new root):

- The basic rotate operation (only used by itself if node is one step away from the root):



- For subtrees of length 3, combine two rotate operations. The complement of the below is given by inverting both trees (take the mirror image of each side):



## 7 Pseudo-Random Numbers

- A **random number generator** is not just a complex program, but a simple one with a mathematical analysis of its properties:
  - Congruential random number generators use the iteration  $x_{i+1} = (Ax_i + B) \bmod C$ .  $A$ ,  $B$  and  $C$  are carefully chosen constants. An initial value  $x_0$  (the seed) must be chosen. The higher order bits appear more random (typically the lowest bit just alternates, but we could divide by 3 to shuffle up the bits).
  - A longer period method uses a recurrence of the form  $a_k = a_{k-b} + a_{k-c}$ .  $b = 31$  and  $c = 55$  works well. This has a state of  $c$  words though, so we need  $c$  elements to get the sequence going (or to restart it).
- A **CRC** (cyclic redundancy check) works by considering a bit sequence to be a polynomial  $M(x)$  (with the bits representing its coefficients). We use another predefined polynomial  $P(x)$  of degree  $n$  (the number of check-bits):
  - Multiply  $M(x)$  by  $x^n$  (right shift  $n$  places) to get  $M'(x)$ .
  - Find the remainder  $R(x)$  of  $M'(x)/P(x)$ .
  - Transmit  $M'(x) + R(x)$ . This is divisible by  $P(x)$ , which is checked at the other end.  $P(x)$  is chosen to reduce the probability of a wrong acceptance.
  - To perform the division in hardware, use a sequence of  $n$  shift registers, with an XOR gate before those with a non-zero coefficient in  $P(x)$ , with extra input being the last shift register. Shift in the value you want to divide (high-order bits first). When the most significant bit in the shift register is set and shifted out, it subtracts  $P(x)$  from the value in the register (as addition and subtraction are equivalent modulo 2). This equates to subtracting some multiple of  $P(x)$  from the input. Once the input has been completely entered, the remainder of dividing by  $P(x)$  is contained in the shift register.

## 8 Data Compression Algorithms

- **Huffman codes** are used to take advantage of the frequency with which a value appears in some data. More frequent values are assigned a smaller code than less frequent values:
  - Set up a frequency table for the symbols in the data (either by knowing the frequencies in advance, or by doing a pass over the data).
  - Take the two items with smallest frequency in the table, and combine them into a two-leaf tree. Put the root of this tree in the table in their place, with frequency being the sum of the two items. Repeat this until the table contains only one item (whose frequency should be the length of the data).
  - Follow the branches down the tree, using ‘0’ for left and ‘1’ for right, to get the Huffman code for each symbol. An alternative is to note the code length for each symbol, and the smallest code for each length, then assign codes (for each length) by incrementing this value.
  - The Huffman table can be compressed by only sending the length of each code (from which the table can be reconstructed). Encode the table using a sequence of nybbles. Starting with the length  $m$  as 0, give the length for each character (e.g. in order

of ASCII code) as a modification of the previous value. So, for example, we have nybbles to represent  $+1, +2, +3, -1, -2, -3$  and  $-4$  to the value of  $m$ . We also have  $Rn$  and  $Zn$  (for  $1 \leq n \leq 4$ ), to represent  $n$  repetitions of the last value of  $m$ , or  $n$  zero-lengths respectively. This can usually get the table under 40 bytes, so it is worthwhile transmitting a new table quite frequently (adaptive encoding).

- **Run-length encoding** represents repeated symbols in the data by the number of times they occur (e.g. AAABBBB becomes 3A4B, or more commonly AA1BB2 to avoid needing extra symbols).
- **Move-to-front buffering** places all the symbols into a vector (the buffer). Encode each symbol encountered in the data as its index into the buffer, then move that symbol to the front of the buffer (displacing the others). This ensures that recently accessed symbols have smaller codes, and frequently used symbols tend to maintain small codes.
- **Arithmetic coding** encodes the entire data as a number in the range  $[0, 1)$ . This range is split into subranges with widths corresponding to the symbol frequencies. If a symbol is selected, that range is then split into sub-ranges of the same proportions. As we descend into subranges, earlier digits become fixed:
  - Using  $n$ -bit arithmetic, we set up a table of values (generally with `eof` at 0), with the proportion representing each symbol given by its frequency. We also create symbol mappings over smaller ranges (greater than half the size of the original).
  - To encode a string, take the first character, and look it up in the portion of the symbol table. If the first bits are the same value, record it. For each of the remaining bits, expand them so that we have another set of  $n$ -bit values to look at (the next bits). Repeat if the first bit is still the same. Use the symbol mapping of the required length to do a look-up of the next symbol. Repeat this for each symbol. If at any stage in order to expand to a valid symbol mapping we would require more than  $n$  bits, it must be the case that the second row of bits are the inverse of the first, so we expand (ignoring this second row), and increment a ‘remember’ counter  $r$ . This may be incremented a number of times, but when the first row of bits is the same, we output it and all the bits we were remembering (its inverse,  $r$  times) and reset  $r$ .
  - To decode, we start off with the full table, and look up the first  $n$  bits of the input in the table, to decode the first character. Looking at that portion of the mapping, if the first bit is the same, we left-shift the input by 1, and expand that section of the mapping (if it isn’t, we don’t shift, and increment a ‘remember’ counter as before). Continue, until the `eof` encoding is encountered, outputting the symbol each time we do a lookup.
- **Lempel Ziv compression** uses an extended alphabet in order to encode sequences of multiple symbols as a single character:
  - To encode, begin with a table of values for the individual symbols. Whenever we encode a symbol, we add that symbol together with the next symbol as a new entry in the table. At each stage, we perform a lookup in the symbol table for the longest entry that encodes that part of the input. As the symbol table grows to size  $2^n$  we change to using  $n + 1$  bits in the encodings.
  - To decode, we can build up the symbol table just by watching which individual symbols are transmitted (if we get A, add an entry for A? and fill it in when we get

the next symbol). We thus always keep track of the symbol table, and know how big it is (thus how many bits are being used for each encoding).

- If the coding table becomes too big, we can restart the entire compression process (e.g. by agreeing the maximum table size on both sides).

- **Burrows-Wheeler block compression** is a method for transforming a string so that it becomes easier to encode:

- Create a matrix of all cyclic rotations of the input, and sort it. Take the last column (which is sufficient to reconstruct the original data) and compact it using run-length encoding, move-to-front buffering, then Huffman coding.
- From the block-transformed message, we know both the last column of the matrix, and the first (since the first column is just the last column sorted). Since each row is a permutation, we simply look at the entry in the first column to find out what follows the entry in the last column in the original message. Furthermore, the first row ending in B corresponds to the same B of the first column starting in B. So, we start at `eof`, which leads us to the first symbol, which we follow to the row that it ends in. Follow this around, until we get back to `eof`.
- We can do the sorting efficiently by firstly doing a radix sort on the first four characters of each permutation. We can then sort individually sets of permutations with the same first four characters.

## 9 Graph Algorithms

- **Warshall's algorithm** is used to find the transitive closure of a relation/graph:

- Consider the graph as an adjacency matrix  $M$ . If we square this matrix, we get an adjacency matrix giving all possible paths of length 2 as non-zero entries. So, the transitive closure is given by  $M^+ = M + M^2 + \dots + M^n$  (where  $n = |V| - 1$ ).
- To find the transitive closure, iterate over all elements in the matrix  $M_{ij}$ . If the element is a ‘1’, then there is a path from vertex  $i$  to vertex  $j$ , so we OR row  $i$  with row  $j$  since  $i$  can get to anything it could previously get to, or that  $j$  can get to. This is  $O(n^3)$ , although the constant factor is small.
- A variant of this finds the minimum cost path between vertices. Using a cost matrix, iterate over all the elements  $C_{ij}$ . If the element is non-zero, then there is some path from  $i$  to  $j$ , so we go through row  $i$  (iterate over  $C_{ik}$ ). Each element is now the minimum of its current value and  $C_{ij} + C_{jk}$ .

- A **strongly connected component** of a graph is a maximal subset of vertices, such that each is accessible from any other. These can be found in  $O(n)$  time:

- Perform a depth-first search on the graph (push a vertex onto the stack, popping it when all paths leading from it have been searched), recording for each vertex  $v$  the discovery time,  $d[v]$  (when it gets pushed onto the stack), and the finishing time  $f[v]$  (when it gets popped off the stack).
- Find the vertex  $v$  that was finished last (and is not already in a strongly connected component). This is a **forefather** –  $\phi(u) = v$  for some vertex  $u$  means that there is a path from  $u$  to  $v$ , and all paths leading out from  $v$  are finished before  $v$  is. We must have discovered  $v$  before any  $u$ , since we finished  $v$  last. Therefore, if there is a

path from  $u$  to  $v$  then there must be a path from  $v$  to  $u$ , or we wouldn't have found  $u$ . So  $v$  and  $u$  are in a strongly connected component.

- The set  $\{u|\phi(u) = v\}$  ( $\{u|u \rightarrow^* v\}$ ) is a strongly connected component. Repeat the above until every vertex is in a strongly connected component.

- A **minimum-cost spanning tree** is a subset of the edges of a graph such that every vertex is part of an edge, and the total cost of these edges is the minimum to give this property (both of these algorithms are greedy, with  $O(n)$  cost):

- **Kruskal's algorithm** – place all vertices into a separate set. Choose the minimum cost edge that forms a bridge between two sets. Repeat this until all sets have been bridged. We need to do a union-find operation to determine if two vertices belong to the same set (i.e. if they have the same root node). As we repeatedly ask this, we can improve efficiency by inserting direct edges from a vertex to the root of its set (to combine two sets, add a pointer from the root of one to the root of the other). To find the minimum edge each time, use a priority queue (heap).
- **Prim's algorithm** – choose a vertex and put it in the set  $S$ . Add to  $S$  the vertex ( $\notin S$ ) that has the least cost connecting it to a vertex in  $S$ . Repeat until all vertices are in  $S$  (remembering the edges we used). This is easily implemented using the adjacency list form, and we can have a pointer in the array of vertices to segregate  $S$  and  $\bar{S}$ .

- **Dijkstra's algorithm** finds the minimum cost path from a vertex  $i$  to all other vertices on the graph:

- Each vertex  $v$  is labelled with the length of the shortest path  $l[v]$  to it, and keeping a set  $S$  of the visited vertices. Initially,  $l[i] = 0$ , and  $l[v] = \infty$  (for  $v \neq i$ ).
- Label all vertices directly reachable from  $i$  with the length of the edge, and add the vertex  $j$  with the smallest label to  $S$ .
- Continue by updating the labels of vertices reachable from  $j$  that are not in  $S$  (only update if  $l[j]$  plus the edge cost is less than its current label), and picking the one with the smallest label to add to  $S$ .
- To find the actual path, backtrack through the labelled graph – if  $j$  is on the path and  $l[j] - e_{jk} = l[k]$  (i.e. there is no slack), then  $k$  is on the path.

- A **bipartite graph** is one in which the vertices are in two distinct sets, such that all edges are between the two sets. A **matching** is a subset of the edges such that each vertex is contained in at most one edge. To find a maximal matching:

- Start off with any matching  $M$ . Find an augmenting path  $P$  (one that starts and ends at nodes not in the matching, with the edges  $e$  alternating between  $e \in M$  and  $e \notin M$ ). Invert the augmenting path, so that  $e \in P \wedge e \notin M \Rightarrow e \in M'$  and  $e \notin P \wedge e \in M \Rightarrow e \in M'$ . Repeat until no more augmenting paths can be found.
- Any non-maximal matching has an augmenting path. For two matchings,  $m$  (non-maximal) and  $M$  (maximal), consider  $G' = m \oplus M$ . This must contain only chains or rings of edges, with alternation between  $m$  and  $M$  (since we don't include shared edges). But there are more edges in  $M$  than in  $m$ , so there must be at least one chain starting and ending with an edge in  $M$  – this is an augmenting path for  $m$ .

## 10 String Algorithms

- A brute force search starts by comparing the pattern with the start of the string, comparing one character at a time. If two characters don't match we fail, so the pattern is shifted right one place, and we continue. Repeat until the pattern is found or the end of the string is reached.
- **Knuth-Morris-Pratt** – construct a table for the pattern giving, for each possible failure position, the amount we can shift the pattern by (e.g. for `xxxAxxxx`, if we fail at position 5 then we can move the pattern 5 spaces, but if we fail at 4, we can only move it by 1, since the failed `A` might have been an `x`). Perform the search in the same was as for brute force, but shift by the appropriate amount at each fail.
- **Boyer-Moore** – this is similar to Knuth-Morris-Pratt, but starts the search from the end of the pattern. Compute the mismatch shift table in the same way (this generally results in bigger shifts). In addition, note down the greatest position of each character in the pattern as the array  $d$ . If we get a mismatch with some character  $c$  in the string at position  $j$  in the pattern, then we must shift by at least  $j - d[c]$  in order for the mismatched character to match the pattern (this is the character shift). Perform the search, and at each failure shift by the maximum of the mismatch and character shifts.
- **Rabin-Karp** – this works by computing a hash of the pattern (of length  $n$ ). As we move through the string, compute the hash of  $n$  digits and compare it to that of the pattern. We only need to perform a comparison if they match. The full hash need only be computed for the first  $n$  characters of the string. For subsequent characters, subtract the contribution of the character no longer being considered, and shift the hash value (multiply by the base) before adding in the new character. This will be performed modulo some value.

## 11 Geometric Algorithms

- The **convex hull** of a set of points is the minimum subset such that they form a polygon that encloses all the other points:
  - Find the lowest left-most point  $O$  (the origin), by seeing which has the smallest  $y$  value (if there is more than one, choose the smallest  $x$  value). This is  $O(n)$ .
  - Sort all the other points by angle relative to  $O$ . Since it is expensive to compute the actual angle, compute an approximation  $t = \frac{y}{|x|+|y|}$  ( $|x| + |y|$  is the Manhattan distance), such that  $\phi_1 > \phi_2 \Leftrightarrow t_1 > t_2$ . Note that no point is to the left of  $O$ . This is  $O(n \log n)$ .
  - Perform a **Graham scan** – go through the sorted list of points. If a point turns left (is to the left of the line joining the previous two points) then add it to the convex hull. Otherwise, if it turns right, delete the previous point and backtrack. When we get back to  $O$  we have completed the convex hull. This is  $O(n)$ .
  - We can use some heuristics to speed this up - e.g. we know that the points with minimum and maximum  $x$ - and  $y$ -coordinates lie on the convex hull, so all the points within the quadrilateral that they form do not lie on the convex hull and need not be considered.
- To find out if a point  $(X, Y)$  lies to the left or right of a line segment  $((x, y), (x', y'))$ , compute  $((X - x)(y' - y) - (x' - x)(Y - y))$ . This is positive if the point is to the left of the line, negative if it lies to the right, and zero if it is on the line.

- To find out if two line segments,  $AB$  and  $CD$ , intersect, we just need to see if  $C$  and  $D$  lie on opposite sides of  $AB$ . If the line segments are finite, we also require that  $A$  and  $B$  be on opposite sides of  $CD$ .
- To find out if a point  $A$  lies within a given polygon, choose a point  $B$  at infinity, and count how many times  $BA$  crosses the boundary lines of the polygon. If there are an odd number of intersections,  $A$  lies within the polygon.
- To find the closest pair of points:
  - Sort all points in order of increasing  $x$ . Split the set of points in two. For each subset (recursively applying the algorithm) find the closest two points – when there are three or less points, compute the distances between each of them to find the closest two.
  - When merging two subsets of points, see which has the closest pair (distance  $\delta$  apart). Go up the boundary line, considering points within  $\delta$  of it. For each point, look at the distance between it and the next  $n$  points along the boundary line (Cormen et al give  $n = 7$ , Richards gives  $n = 3$ ).
  - We can do this efficiently if we return not only the closest pair in a subset, but that subset sorted in increasing  $y$ . The merge operation is then performed in linear time.