CS 240 — Data Structures and Data Management

Kevin James

Spring 2014

Contents

1 Algorithms

An **algorithm** is a step-by-step process for carrying out a set of operations given an arbitrary problem instance. An algorithm **solves** a problem if, for every instance of the problem, the algorithm finds a valid solution in finite time.

A **program** is an implementation of an algorithm using a specified programming language.

For each problem we can have several algorithms and for each algorithm we can have several programs (implementations).

In practice, given a problem:

1. Design an algorithm.
2. Assess the correctness of that algorithm.
3. If the algorithm is acceptable, implement it. Otherwise, return to step 1.

When determining the efficiency of algorithms, we tend to be primarily concerned with either the runtime or the memory requirements. In this course, we will focus mostly on the runtime.

To perform runtime analysis, we may simply implement the algorithm and use some method to determine the end-to-end time of the program. Unfortunately, this approach has many variables: test system, programming language, programmer skill, compiler choice, input selection, ... This, of course, makes manual implementation a bad approach.

An idealized implementation uses a **Random Access Machine (RAM)** model. RAM systems have constant time access to memory locations and constant time primitive operations, thus the running time is determinable (as the number of memory operations plus the number of primitive operations).

We can also generally use **order notation** to compare multiple algorithms. For the most part, we compare assuming n is very large, since for small values of n the runtime will be miniscule regardless of algorithm.

A timing function is a function T_A such that $T_A : \{Q\} \to \mathbb{R} > 0$. We denote the runtime of a function as $T(f(x))$, for example: $T(3 \times 4)$ may be equal to $0.8ns = 8\text{ops}$. The return value is the number of operations required in the worst-case scenario.

Example: given $T_A(n) = 1000000n + 2000000000$ and $T_B(n) = 0.01n^2$, which is 'better'? For $n < 100000000$, algorithm B is better. Since we only care about large inputs, though, we say A is better overall.

Definition 1.1. *$f(n) \in O(g(n))$ if there exists a positive real number c and an integer $n_0 > 0$ such that $\forall n \geq n_0$, $f(n) \leq cg(n)$.*

Example 1.1. *Prove that $2010n^2 + 1388 \in O(n^3)$.*

Proof. We want to prove that for all positive $c \in \mathbb{Z}$, there exists some n_0 such that for all $n \geq n_0$, $2010n^2 + 1388 \leq cn^3$. So for $n > 1388$ we have $2010n^2 + 1388 \leq 2011n^2 \leq cn^3$ which in turn gives us $2011n^2 \leq cn^3 \iff 2011 \leq cn$. Then this holds given $n_0 = 2011$. □

More concretely, we can say that $f(n) \in O(af(n))$ and $bf(n) \in O(f(n))$. It's also worth noting that order notation is transitive (e.g. $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ implies $f(n) \in O(h(n))$).

We use five different symbols to denote order notation:

- o denotes a function *always less* than a given order
- O denotes a function *less than or equal* to a given order
- Θ denotes a function *exactly equal* to a given order
- Ω denotes a function *greater than or equal* to a given order
- ω denotes a function *always greater* than a given order

More concretely, we have the formulae
Definition 1.2 (o)**.** *$f(n) \in o(g(n))$ if for all $c > 0$ there exists a constant $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$.*
Definition 1.3 (O)**.** *$f(n) \in O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.*
Definition 1.4 (Θ)**.** *$f(n) \in \Theta(g(n))$ if there exists constants $c_1, c_2 > 0$ such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.*
Definition 1.5 (Ω)**.** *$f(n) \in \Omega(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.*
Definition 1.6 (ω)**.** *$f(n) \in \omega(g(n))$ if for all $c > 0$ there exists a constant $n_0 > 0$ such that $f(n) > c \cdot g(n)$ for all $n \geq n_0$.*
Example 1.2. *For the psuedo-function*

```
function(n):
    sum = 0
    for i=1 to n:
        for j=i to n:
            sum = sum + (i-j)^2
            sum = sum^2
    return sum
```

we find the order equation

$$\begin{aligned}
\Theta(1) + \sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(1) + \Theta(1) &= \Theta(1)\sum_{i=1}^{n}\sum_{j=1}^{n}1 \\
&= \Theta(1)\sum_{i=1}^{n}(n-i+1) \\
&= \Theta(1)\left(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i + \sum_{i=1}^{n}1\right) \\
&= \Theta(1)\left(n^2 + i^n + n\right) \\
&= \Theta(n^2) + \Theta(i^n) + \Theta(n) \\
&= \Theta(i^n)
\end{aligned}$$

Example 1.3. *For the psuedo-function*

```
function(n):
    sum = 0
    for i=1 to n:
        j = i
        while j >= 1:
            sum = sum + i/j
            j = j/2
    return sum
```

we find the order equation

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{\log_2 i} c &= \sum_{i=1}^{n}(c\log_2 i) \\
&= c\big(\log 1 + \log 2 + \log 3 + \cdots + \log n\big) \\
\textit{all } n \textit{ of our terms are below } \log n & \qquad \textit{half of our } n \textit{ terms are above } \frac{n}{2} \\
= O(n\log n) & \qquad = \Omega(\frac{n}{2}\log\frac{n}{2}) \\
& \qquad = \Omega(n\log n) \\
& = \Theta(n\log n)
\end{aligned}$$

Since it is too hard to describe the runtime for every possible input, we decide to only describe the worst case behaviour. The worst case gives us a *guarantee* for required completion time and tends to describe most cases.

More formally, we take

$$\max_I\{T_A(I)\}$$

2 Data Types

2.1 Stacks

A **stack** is a collection of items which supports the operations `push` (insert an item), `pop` (remove the most recently inserted item), `peek` (view the last item), and `isEmpty`.

The most common ways to implement a stack are as a

- **linked list**: a pointer to the top of the stack is maintained and is moved whenever an item is inserted or removed, or an
- **array**: the last item is easy to access, though sometimes resizing the array will be necessary

2.1.1 Dynamic Arrays

Linked lists support $O(1)$ insertion and deletion, $O(n)$ accessing. Arrays are vice-versa. **Dynamic arrays** offer a compromise: $O(1)$ for both, but can only insert or delete from the end of the list.

2.2 Queues

A **queue** is a data structure where you `insert` at the end of the list and `dequeue` from the front. Implementations are the same as a stack: either an array or a linked list may be used, though a pointer or reference to the first item is necessary.

2.2.1 Priority Queues

A **priority queue** is similar to a queue, but each element has a priority attached to it (a numerical "score"). The elements are dequeue'd in order of priority. A priority queue supports

- `insert(x, p)`: inserts an element x with priority p
- `delete()`: deletes the element with the highest priority p
- `peek()`: views the top element

One of the most useful applications of priority queues is for sorting: by inserting all elements from an array and then deleting them, we will have the elements returned in a correctly sorted order. Then we simply need to examine the runtime efficiency of `insert` and `delete` to determine the speed of our sorting algorithm.

There are two common implementations of priority queues: unsorted arrays and heaps.

In an **unsorted array**, insertion takes $O(1)$ time, since the element is simply placed at the end of the array. Deletion must walk the array, then replace the deleted (read: smallest it has seen as it walked) with the last element in the array. Thus, this is $O(n) + O(1) = O(n)$ time. Sorting, then, takes $O(n^2)$ time.

A better method is to use heaps.

2.3 Heaps

A **heap** is a *complete* binary tree with the *heap property*. To have a complete binary tree, the tree must be structured such that for each node in the tree that mode must have zero or two children, unless that node is the rightmost leaf in the bottom level (in which case it may have only one). Furthermore, deeper leaves must be leftmost in the tree and all the leaves in the bottom two levels must be consecutive.

The **heap property** is the principle such that all nodes have a lower priority than that of their parent. In the case of multiple elements with the same priority, arbitrary element positioning is possible.

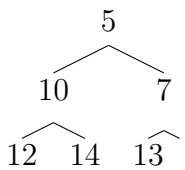

Figure 1: A min-PQ tree that satisfies the heap properties

This is not a binary search tree, since the larger element does not need to be on the right. In fact, there is no relationship of any kind with a node's siblings—the only relationship is with its parent.

When we remove an item from a heap, we must replace that item with the current lowest element. This will guarantee that the tree remains in a complete form (that all holes are filled). After replacing it with this element, the newly moved element must be sorted: recursively compare its priority with that of its children; if any children have a higher priority, it should swap with its parent. Otherwise, stop recursing.

When inserting, we use the opposite process: the new element is placed at the base of the tree, then it is recursively compared and potentially swapped with its parent until it is in a valid position.

Note that we do not implement tree-like data structures as such, but rather as arrays. The array follows the format such that the tree represented in figure 1 would be placed in an array as $[5, 10, 7, 12, 14, 13]$. More concretely: the parent of a node at index n is at index $\lfloor \frac{n-1}{2} \rfloor$. Its children are located at indices $2n + 1$ and $2n + 2$.

Heap functions have worst case running times bounded as follows

- Heap insertion: $\Theta(\log n)$
- Heap deletion: $\Theta(\log n)$
- Top-down heap creation: $\Theta(n \log n)$
- Bottom-up heap creation: $O(n)$

2.3.1 Heap Insert

The item to-be-inserted must be placed in the only possible location to preserve integrity: i.e. the bottom level in the first available position. However, this may violate heap ordering procedure; we must perform bottom-up heap swaps until the ordering is satisfied.

To perform a bottom-up heap swap, we must

- compare node with parent
- if the node is larger than the parent
  - swap
  - recurse
- else
  - quit

These swaps can be performed in $\Theta(1)$ time and since the maximum number of swaps is equal to the height of the heap we have an overall time complexity of $\Theta(\log n)$. This worst case would occur when the item to-be-inserted is larger than anything else on the heap, since it would need to swap all the way to the top.

2.3.2 Heap Delete

The item to-be-deleted can simply be removed form our heap; however, we must replace this vacated position with the lowest item in our heap. This satisfies the structural integrity of the heap. Afterward, we must follow top-down heap swapping in order to satisfy ordering integrity.

To perform a top-down heap swap, we must

- compare node with children
- if the node is smaller than at least one child
  - swap with the largest child
  - recurse
- else
  - quit

Again, these swaps can be performed in $\Theta(1)$ time and we can perform no more than $\lfloor \log n \rfloor$ swaps. As such, we have worst case time complexity $\Theta(\log n)$ when the deleted item is at the top of the heap and the replacement item is the smallest in the heap.

2.3.3 Heap Creation

Given an array in no particular order, our goal is to adjust the ordering such that we have a valid heap.

The obvious approach is to call the heap insertion method once for each item in our input array. This is the **top-down** approach to heap creation.

This can generally done in place, as we can treat the input array as a structurally valid but orderingly incorrect binary tree. As such, each insertion operation simply 'ignores' values to its right. Since we are calling our $\Theta(\log n)$ insertion method n times, we see that top-down heap creation has runtime complexity $\Theta(n \log n)$.

More formally, we see we have an upper bound such that the cost of the swaps is proportional to the depth of the node (i.e. which level it is on) and the number of nodes to process. This depth is given by $O(\log n)$ and the number of nodes by n, so we have $O(n \log n)$ as our upper bound. Our lower bound is given as such: the worst case occurs when the input array is sorted in ascending order (i.e. requires we perform the maximal number of swaps for each insertion). In this case, we would be required to swap $\lfloor \log n \rfloor$ times for each of our n insertions.

Lemma 2.1. *At least half of the nodes in a heap are in the last two levels (have depth $d \geq h - 1$)*

Based on lemma 2.1, we can ignore all levels above the bottom two when computing the lower bound. The worst case number of swaps at level $h - 1$ or h is at least $h - 1$, thus the overall number of swaps $s \geq \frac{n}{2}(h - 1)$ which is $\Omega(n \log n)$.

We can also creation a heap in the **bottom-up** fashion as such: the entire array is transformed into a heap in one iteration, by finding the elements from largest to smallest and inserting them in that order.

We begin in the same way as for top-down creation, by placing our array into a incomplete binary tree (or associating our already-made array with a binary tree). Now, we start with the element in the last position and perform a top-down swap operation for each element in the array (recursing backwards).

This is very similar to the top-down heap creation method, but has different boundaries.

Since our swaps can now be made in the opposite direction, our lowest level of the tree no longer needs to swap and our second lowest level can only possible swap once. As such, over half of our tree has a maximal swapping runtime of $O(1)$ and the remainder of the tree has $O(\log n)$, whereas the top-down approach uses the constant boundary $O(\log n)$.

More concretely, the total number of swaps we can perform is equal to

$$\begin{aligned}\sum_{i=0}^{h} i2^{h-i} &< n\sum_{i=0}^{\infty}\frac{i}{2^i}\\ &< 2n\end{aligned}$$

and thus we have the upper bound of $O(n)$.

The lower bound is at least $\Omega(n)$ (since we must run our heap creation on n elements), thus we have an overall runtime complexity of $\Theta(n)$.

3 Sorting

There exist many sorting algorithms:

Algorithm	Time Complexity	Stable	In Place	Memory Complexity
Selection Sort	$\Theta(n^2)$	Maybe	Yes	$O(1)$
Insertion Sort	$\Theta(n^2)$	Yes	Yes	$O(1)$
Merge Sort	$\Theta(n \log n)$	Yes	No	$O(n)$
Heap Sort	$\Theta(n \log n)$	No	Yes	$O(1)$
Quick Sort	$\Theta(n \log n)$	Maybe	Maybe	$O(\log n)$ or $O(n)$
Counting Sort	$\Theta(n + k)$	Yes	No	$O(n + k)$
Radix Sort Sort	$\Theta(d(n + k))$	Yes	No	$O(n + k)$

For a comparison-based algorithm, we have a lower bound of $\Omega(n \log n)$. For non-comparison-based algorithms, we can achieve $O(n)$ ($O(n + k)$ and $O(d(n + k))$ can be made to be $O(n)$ with a proper selection of k and d).

3.1 PQ Sort

As mentioned above, we can sort data using a priority queue. Both `insert` and `delete` take $O(\log n)$ time, as does recursively sorting any newly moved elements. Thus, for sorting n elements we see that PQ sort will take $O(n \log n)$ time, which is the same as mergesort.

3.2 Heap Sort

Heap sort is a specialized PQ sort.

The heap sort algorithm is

```
heapSort(A, n):
    H = new Heap(n)
    for i in range(0, n - 1):
        heapInsert(H, A[i])
    for i in range(0, n - 1):
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

or given our "heapify" algorithm

```
heapSort(A, n):
    H = heapify(A, n)
    for i in range(0, n - 1):
        A[n-1 - i] = heapDeleteMax(H, n-1)
```

which gives us a complexity of

$$\begin{aligned} O(n) + \sum_{i=0}^{n-1} \Theta(\log n) &= O(n) + \Theta(n \log n) \\ &= \Theta(n \log n) \end{aligned}$$

3.3 Quick Sort

The quick sort algorithm works by having one function which selects the optimal pivot point, moving elements such that all items left of the pivot are smaller and vice-versa, then recursing on each side.

3.3.1 Selecting

Given an array A, we want to find the kth smallest (or largest) element. We do this by recursively taking the first element (the **pivot**), then sorting the elements such that any elements smaller than the pivot are on its left (and vice versa). This continues until there are fewer than k elements on the requested side.

```
quickSelect(A, k):
    p = randomPivot(A)
    i = partition(A, p)
    if k < i:
        quickSelect(A[0, i-1], k)
    elif k > i:
        quickSelect(A[i+1, n-1], k - i - 1)
    else:
        return p
```

When selecting, it is only required that we recurse on one side of the pivot (i.e. the side which contains the element we are searching for).

The partition function is defined thusly

```
partition(A, p):
    swap(A[0], A[p])
    i = 1
    j = n - 1
    while true:
        while i < n and A[i] < A[0]:
            i = i + 1
        while j >= 1 and A[j] > A[0]:
            j = j - 1
        if j < i:
            break
        else:
            swap(A[i], A[j])
```

In the *worst case*, we have $\Theta(n^2)$ time. This would occur if the selected pivot is always $\max(A)$ and we are searching for the smallest element, for example in the array $[n, n-1, n-2, \ldots, 3, 2, 1]$.

In the *average case*, the probability of selecting a "good" pivot is 50%. After r recursive calls, this means the array was halved approximately $\frac{r}{2}$ times. After $4 \log n$ calls, we must be left with an array of size 1. Thus we have

$$\begin{aligned}
f(n) &\leq \frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn \\
&\leq 2cn + 2cn\left(\frac{3n}{4}\right) + 2c\left(\frac{9n}{16}\right) + \cdots + T(1) \\
&\leq \frac{1}{2}\left[\frac{1}{2}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + cn\right] + \frac{1}{2}\left[\frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}T\left(\frac{9n}{16}\right) + c\frac{3n}{4}\right] \\
&\leq \frac{1}{4}T(n) + \frac{1}{2}T\left(\frac{3n}{4}\right) + \frac{1}{2}cn + \frac{1}{4}T\left(\frac{9n}{16}\right) + c\frac{3n}{8} + cn \\
&\leq T(1) + 2cn\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i \\
&= \Theta(n)
\end{aligned}$$

3.3.2 Sorting

The general quick sort algorithm is

```
quickSort(A, n):
    if n == 0:
        return
    p = randomPivot(A)
    i = partition(A, p)
```

```
    quickSort(A[0, i-1])
    quickSort(A[i+1, n])
```

The worst case is $\Theta(n^2)$ and the best case is $\Theta(n \log n)$. Fortunately, the worst case is extremely unlikely and the average case is also $\Theta(n \log n)$.

3.4 Counting Sort

Assume each element e in an array satisfies $0 \leq e \leq k-1$ for some known value k. Then we can use the **counting sort** algorithm on this array, implemented as follows:

```
countingSort(A[1..n], k):
    C = [0]*k
    for i = 0 to n-1:
        C[A[i]]++
    for i = 1 to k-1:
        C[i] += C[i-1]
    B = copy(A)
    for i = reverse(0 to n-1):
        C[B[i]]--
        A[C[B[i]]] = B[i]
```

Counting sort has time complexity $\Theta(n+k)$, space complexity $\Theta(n+k)$, and is a **stable** algorithm (equal items are not reordered).

3.5 Radix Sort

Assume we have a set of (probably) unique integers. Then we can use the **radix sort** algorithm on this set, implemented as follows:

```
radixSort(A[1..n], d, k):
    for i = 0 to d - 1:
        countingSort(A, k) //with each x_i as the key
```

We can perform radix-sort from either the left or right sides (LSD vs. MSD). The MSD-Radix sort, though, can only be performed if all integers have the same number of digits (or are padded with zeroes).

Radix sort has time complexity $\Theta\big(d(n+k)\big)$ and space complexity $\Theta(n+k)$.

4 Dictionaries

A **dictionary** ("associative array") is a collection of items, each of which contain a **key** and some **data** and is referred to as a **key-value pair**. Keys can be compared and are (typically) unique.

We define the following functions for a dictionary:

- `search(k)`
- `insert(k, v)`
- `delete(k)`
- and optionally: `join`, `isEmpty`, `size`, ...

We can implement a dictionary in several ways

- as an unordered array: has $\Theta(n)$ search and deletion time and $\Theta(1)$ insertion time
- as a sorted array: has $\Theta(\log n)$ search time and $\Theta(n)$ insertion and deletion time
- as a linked list: has $\Theta(n)$ search and deletion time and $\Theta(1)$ insertion time

4.1 Hashing

One problem with dictionaries is that we can only address directly if the keys are integers. If we use a **hashing function** $h : U \to \{0, 1, \ldots, M-1\}$ (assuming all keys come from some universe U), we find M unique keys which we can use for addressing. Unfortunately, h is not injective and we may have many keys mapping to the same integer.

A **hash table dictionary** has an array T of size M (the hash table). An item with key k is stored in $T[h(k)]$, and we can have `search`, `insert`, and `delete` all cost $O(1)$. The challenges to this approach are selecting a good hashing algorithm and dealing with **collisions** (when $h(k_1) = h(k_2)$).

There are two common hashing functions (assuming all keys are integers or can be mapped to integers):

- the **division method** $h(k) = k \pmod M$ where M is a prime not close to a power of two, and
- the **multiplication method** $h(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$ from some constant floating-point number $0 < A < 1$. Knuth suggests $A = \frac{\sqrt{5}-1}{2}$.

Even the best functions may have collisions. There are two basic strategies for dealing with the case where we are attempting to insert an item into an already existing location. We could allow multiple items at each location (buckets) or allow each item to go into multiple locations (open addressing). The average cost of collision operations is denoted by the **load factor** $\alpha = \frac{n}{M}$ and we will generally want to rebuild our entire hash table (i.e. rehash) when the load factor gets either too large or too small. This should cost roughly $\Theta(n + M)$.

The expected time of the first collision is after $\sqrt{\frac{\pi M}{2}}$ insertions, where M is the size of the table. MH_M items must be inserted before each slot in the table must have at least one item, where H_M is roughly $\log M$.

4.1.1 Chaining

To use **chaining** to solve our collisions, each table entry must be a bucket containing zero or more KVPs. This could be implemented with any dictionary (or recursive hashtable). The simplest

approach is an unsorted linked list in each bucket.

Assuming we have a uniform hashing function, the average size of each bucket is $\alpha = \frac{n}{M}$. Then we have

- `search` has $\Theta(1+\alpha)$ average-case time and $\Theta(n)$ worst-case time
- `insert` has $O(1)$ worst-case time
- `delete` has $\Theta(1+\alpha)$ average-case time and $\Theta(n)$ worst-case time

If we maintain $M \in \Theta(n)$, the average cost for each of these functions is $O(1)$. This can be accomplished by rehashing when $n < c_0 M$ or $n > c_1 M$ for some $0 < c_0 < c_1$.

4.1.2 Open Addressing

We can also use **open addressing** to solve our collision problems. In this case, each hash table entry holds only one item but any key k can be placed in multiple locations. `search` and `insert` follow a **probe sequence** of possible locations for key k: $h(k,0), h(k,1), h(k,2), \ldots$. `delete` becomes problematic: we must distinguish between *empty* and *deleted* locations. The simplest idea is lineary probing: $h(k,i) = (h(k) + i) \pmod M$.

Theorem 4.1. *With linear probing, the average number of probess required to search in a hash table of size M with α load factor $[n = \alpha M$ keys$]$ is $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ when successful and $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ when unsuccessful.*

If the average cost of searches is t, then for we set $\alpha < 1 - \frac{1}{\sqrt{t}}$.

4.1.3 Double Hashing

Suppose we have two hash functions h_1 and h_2 and that these functions are independant. Then we can define $h(k,i) = h_1(k) + ih_2(k) \pmod m$. Our three standard functions work in the same way as linear probing, but follow this different probe sequence.

In this case, we avoid creating large islands of numbers which must be repeatedly walked through.

Theorem 4.2. *With double hashing, the average number of probess required to search in a hash table of size M with α load factor $[n = \alpha M$ keys$]$ is $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$ when successful and $\frac{1}{1-\alpha}$ when unsuccessful.*

If the average cost of searches is t, then for we set $\alpha < 1 - \frac{1}{t}$.

4.1.4 Cuckoo Hashing

The Cuckoo hashing algorithm also uses two hashing algorithms: the idea, though, is to always insert a new item into $h_1(k)$. If this overwrites another element, we simply re-insert the old element into its alternate position.

The "alternate position" is defined as follows: the two hash functions are used to create two completely separate hash tables. Thus, each element can exist in any one of two possible **nests**.

This gives us $O(1)$ search and deletion as well as amortized $O(1)$ insertion, though it does require mostly random hash functions.

4.2 Memory Usage

If we have a very large dictionary which must be stored in external memory, we want to minimize the number of page read/writes and faults.

Linear probing generally requires each hash table to be on the same page; thus, each α must be small and we find ourselves wasting space. We also find ourselves commonly rehashing an entire table.

4.2.1 Extendible Hashing

If external memory is stored in blocks of size S, our goal should be to access as few as possible. Similar to a B tree of height 1 and max leaf-size S, we can do the following:

We store the **directory** (root node) in internal memory. This directory contains a hashtable of size 2^d, where d is the order. Each directory points to a block stored in external memory containing at most S items, sorted by hash value.

To search for a key k in the dictionary, we find block B with the first d bits of $H(k)$ as such: $\lfloor \frac{h(k)}{2^{L-d}} \rfloor$. Then, we simply perform a binary search in block B for our hash value. This algorithm is $\Theta(\log S)$ and generates one page fault.

To insert into this dictionary, we search for $h(k)$ to find the proper block for insertion. IF the block has space, we insert our key. Otherwise, we perform a **block split**: we separate B into B_0 and B_1 where the items are split by the k_Bth bit, then update our dictionary references. Note that if our dictionary does not have enough space for a split, we must first double its size and update references accordingly (this is called a **dictionary grow**).

To delete from this dictionary, we perform the reverse operation from insertion: we search for block B and remove k from it. If n_B is too small, we perform a **block merge**, and if every block B has local depth $k_B \leq d - 1$, we perform a **dictionary shrink**.

Both insertion and deletion can be performed in $\Theta(S)$ time, though a dictionary grow/shrink takes $\Theta(2^d)$ time. These operations generate one or two page faults, depending on whether there is a block split/merge.

4.3 Balanced Search Trees

We can use either hash tables or balanced search trees for these types of operations. The advantages are as follows:

BSTs:

- $O(\log n)$ worst-case time

- no assumptions, special functions, or known input proportions
- no wasted space
- never need to rebuild entire structure

Hash Tables:

- $O(1)$ average cost
- flexible load factor parameters
- Cuckoo hashing achieves $O(1)$ worst-case for search and delete

Note that both approaches can be adapted to minimize page faults.

4.4 Multi-Dimensional Data

An item is said to be multi-dimensional if it has $d > 1$ aspects (co-ordinates). Each item, then, corresponds to an item in d-dimensional space.

Though we can use ordered arrays for one dimensional range searches, we prefer a balanced BST (e.g. AVL tree) when generalizing to higher dimensions. Using a balanced tree, we can perform a range search in $O(\log n)$ time and output the answers in $O(k + \log n)$, where k is the number of answers.

5 Trees

5.1 Binary Trees

Binary trees are the most common tree structure. Their defining factor is nodes with a single value.

5.1.1 Binary Search Trees

A **BST** is either empty or contains a KVP, left-child BST, and right-child BST. Every key k_L in $T.left$ is less than the root key, every key k_R in $T.right$ is greater. These properties are recursive.

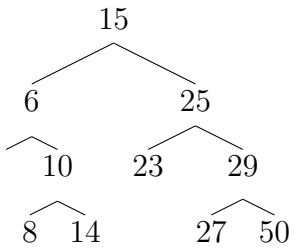

Figure 2: A sample binary search tree

`search`, `insert` and `delete` all have time complexities $\Theta(h)$, where h is the height of the tree. h has a best-case value of $\log n$ and a worst-case value of n.

There exist the following BST variations:

- Randomized BST
- Self-Balancing BST
- Threaded BST

We can perform a single-dimensional range search on a BST as:

```
BST-rangeSearch(T, k1, k2):
  if T = nil:
    return
  if key(T) < k1:
    BST-rangeSearch(T.right, k1, k2)
  if key(T) > k2:
    BST-rangeSearch(T.left, k1, k2)

  if k1  key(T)  k2:
    BST-rangeSearch(T.left, k1, k2)
    report key(T)
    BST-rangeSearch(T.right, k1, k2)
```

Given k is the number of reported items, this algorithm runs in $O(k + \log n)$ time.

5.1.2 AVL Trees

An **AVL tree** is a BST with an additional structural property: the heights of the left and right subtree must differ by no more than one (where an empty tree is defined to have a height of -1).

At each non-empty node, we store height$R$ − height$(L) \in \{-1, 0, 1\}$: -1 means the tree is left-heavy, 1 means the tree is right-heavy, and 0 means the tree is balanced. We could store the actual height, but storing the balances is simpler and more convenient.

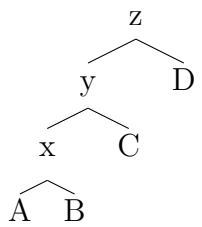

Figure 3: An inbalanced AVL tree.

An AVL tree can perform **rotations** to rebalance itself.

This function can be represented as

```
rotate_right(T):
    newroot = T.left
```

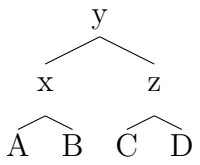

Figure 4: A balanced AVL tree (after rotating right).

```
    T.left = newroot.right
    newroot.right = T
    return newroot

rotate_left(T):
    newroot = T.right
    T.right = newroot.left
    newroot.left = T
```

We can also perform **double rotations**, which may be either two of the same rotation operation or one of each.

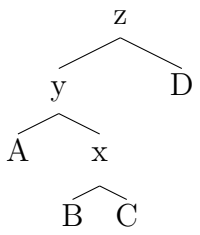

Figure 5: An inbalanced AVL tree requiring two rotations (left then right).

The `search` operation is the same as a BST and is in $\Theta(h)$ time. The `insert` uses rotations and will thus be $\Theta(h)$. Finally, the `delete` operation swaps then applies fixes (rotations) and is thus $\Theta(h)$.

We define $N(h)$ to be the least number of nodes in an AVL with height h. One subtree must have height of at least $h-1$, the other at least $h-2$. Then we have

$$N(h) = \begin{cases} 1 + N(h-1) + N(h-2) & h \geq 1 \\ 1 & h = 0 \\ 0 & h = -1 \end{cases}$$

Red-black trees are another binary tree implementation which is somewhat similar to AVL trees. They have a minsize 1 and a maxsize 3, but represent 2- or 3-nodes as two or three distinct binary nodes with a color (i.e. "red" or "black").

5.2 B Trees

A **B tree** is a classification of trees which have a more generalized structure than BSTs. Instead of each node having a single value, B tree nodes have multiple values. All B tree leaves must be

at the same level.

A **B tree of minsize** d is a B tree with each node containing at most $2d$ values and each non-root node containing at least d values. A B tree of minsize d is typically also refered to as a B tree of order $(2d + 1)$, or (even more specifically) a $(d + 1)$-$(2d + 1)$ tree. For example, a 2-3 tree (introduced below) has $d = 1$.

5.2.1 2-3 Trees

A **2-3 tree** is a BST with additional structural properties:

- every node either contains one KVP and two children or two KVPS and three children.
- all the leaves (nodes without children) are at the same level

Searching through a 1-node is the same as a BST, but we must examine both keys in a 2-node to determine the most accurate path.

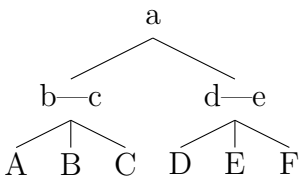

Figure 6: A balanced 2-3 tree.

To insert into a 2-3 tree, we first search to find the leaf where the key belongs. If the leaf has a single KVP, we simply add the new key and make a 3-node. Otherwise, we sort the keys, split the left-most and right-most into two 1-nodes, and recursively insert the middle key into the parent (along with the link).

To delete, we first swap the KVP with its successor so we are always deleting from a leaf. If the target node is a 2-node, we simply delete the key. Otherwise, if the target node has an immediate 2-node sibling we perform a **transfer** (put the intermediate KVP in the parent between the two nodes into the target node and replace it with the adjacent KVP in the sibling node). Otherwise, we **merge** the target node and a 1-node sibling by removing the target node and recursively deleting the intermediate KVP in the parent and adding it the the sibling node.

Note that the 2-3 tree `search`, `insert`, and `delete` implementations defined above are the same for all B tree implementations.

Each implementation of a B tree (including those not mentioned) has the following variations

- B Trees: standard implementation
- B+ Trees: all data stored in leaves (nodes are only used for determining which path to follow)
- B# Trees: all data stored in leaves, rotation operation is only defined for siblings
- B* Trees: all data stored in leaves, nodes are kept $\frac{2}{3}$ full by redistributing to three binary children leaves (two with data, one without)

5.3 Range-Finding Trees

When we find ourselves needing to perform a multi-dimensional range search, we have several options:

- reduce our data to a single key (this is not simple or straightforward)
- use a dictionary for each dimension (inefficent, wastes space)
- use partition trees
- use multi-dimensional range trees

5.3.1 Partition Trees

5.3.1.1 Quadtrees can be used to improve the efficiency of our range searches. If we have n points in $P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, we can build a quadtree as follows:

1. Find a square R which contains all the points in P (using minimum and maximum x and y values)
2. Let the root of the quadtree correspond to R
3. We partition (split) R into four equal subsquares (quadrants), each of which will correspond to a child of R
4. Recursively repeat the previous step for each node containing more than one point
5. Delete all leaves which do not contain a point

We can search a quadtree in the same way as a BST. When we insert, we simply place the item in the correct node and split that node if it contains more than one point. To delete, we search for the point, delete it, and walk back up the tree to delete unnecessary splits.

We can perform a range search on a quadtree as:

```
QTree-RangeSearch(T, R):
  if (T is a leaf):
    if (T.point  R):
      report T.point
  for each child C of T:
    if C.region intersect R is null:
      QTree-RangeSearch(C, R)
```

The complexity of a quadtree range search is $O(nh)$, no matter the answer (e.g. including $\varnothing$). We also define the **spread factor** of points P: $\beta(P) = \frac{d_{\max}}{d_{\min}}$, where $d_{\max}$ and $d_{\min}$ are the maximum or minimum distance between two points in P.

Note the height of a quadtree is $h \in \Theta(\log_2 \frac{d_{\max}}{d_{\min}})$.

The initial build time complexity of a quadtree is $\Theta(nh)$.

Quadtrees are easy to compute and deal with, but can waste a large amount of space and can have large heights (depending on distribution of points). Furthermore, they are simple to generalize to higher dimensions (e.g. octrees).

5.3.1.2 The idea behind **KD trees** is to split the points into two roughly equal sides and thus create a balanced binary tree. We do this by alternating between vertical and horizontal "splitting lines" until each section has only one point. This gives us a time complexity of $\Theta(n \log n)$ and the height of the tree $h \in \Theta(\log n)$.

The range search algorithm

```
def KDTree-rangeSearch(T,R):
  if T.empty:
    return
  if T.point in R:
    report T.point
  for each child C in T:
    if C.region intersect R is null:
      KDTree-rangeSearch(C, R)
```

has a complexity of $\Theta(k + U)$, where k is the number of keys reported and U is the number of regions we unsuccessfully visit.

We define $Q(n)0$ as the maximum number of regions which intersect a splitting line. We have $Q(n) = 2Q(\frac{1}{4}n) + O(1)$ which solves to $Q(n) \in O(\sqrt{(}n))$ and we have overall time complexity of range search given as $O(k + \sqrt{(}n))$.

For higher dimensional KD trees, we have storage space $O(n)$, construction time $O(n \log n)$, and range search time $O(n^{1-\frac{1}{d}} + k)$ (where d is the number of dimensions and is considered constant).

5.3.2 Range Trees

A **range tree** is a tree of trees (a multi-level data structure). In essence, a range tree is built by creating a balanced BST determined solely based on the first-dimension co-ordinates of each element. Then, for every node in this tree, a *new* balanced BST is created determined by the second-dimension co-ordinates of the nodes below the selected node in the current tree.

A range search is simply a range search in the outer-most tree which then recurses on each "top" node in range. Thus for each dimension, we perform a standard range search on that tree level number (e.g. the trees created using the nodes of the outer-most tree are used to search with second-dimension data).

In two dimensions, a range tree search gives run time $O(k + \log^2 n)$. Note that construction time and storage usage both are in $O(n \log n)$. This generalizes to:

- storage space: $O(n \log^{d-1} n)$
- construction time: $O(n \log^{d-1} n)$
- range search time: $O(k + \log^d n)$

6 String Matching

6.1 Pattern Matching

The act of **pattern matching** is to search for a given string in a large body of text. We refer to $T[0 \dots n-1]$ as the **haystack**—the text being searched, and $P[0 \dots m-1]$ as the **needle**—the pattern we are searching for. Our goal is to return the first i such that

$$P[j] = T[i+j]$$

This is the first occurence of P in T.

We define the substring $T[i \dots j]$ as a string of length $j-i+1$ consisting of characters $T[i], T[i+1], \dots, T[j-1], T[j]$ in order. We also define the **prefix** $T[0 \dots i]$ and the **suffix** $T[i \dots n-1]$ where $0 \leq i \leq n-1$.

Pattern matching algorithms consist of guesses and checks:

- A **guess** is a position i such that P might start at $T[i]$. Valid guesses obey $o \leq i \leq n-m$.
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $P[j]$ to $T[i+j]$. We must perform m checks for a correct guess, though we may make fewer checks to determine an incorrect guess.

We can find the brute force algorithm as:

```
bruteForce(T[0..n-1], P[0..m-1]):
  for i = 0 to n - m:
    match = true
    j = 0
    while j < m and match:
      if T[i+j] == P[j]:
        j = j + 1
      else:
        match = false
    if match:
      return i
  return FAIL
```

The worst possible input to this algorithm is $P = a^{m-1}b, T = a^n$. This gives us a worst case time complexity of $\Theta\big((n-m+1)m\big)$ or, if $m \leq \frac{1}{2}n$, $\Theta(nm)$.

Fortunately, we can find more sophisticated algorithms. More specifically, **KMP** and **Boyer-Moore** both perform extra preprocessing on P and eliminate guesses based on completed matches and mismatches.

6.2 KMP

The **Knuth-Morris-Pratt** algorithm matches text in left-to-right. It also shifts the pattern more intelligently than the brute force algorithm.

If a check fails (a **mismatch** occurs), we can shift the pattern at most x spaces to the right, where x is the largest prefix of $P[0 \ldots j]$ which is also a suffix of $P[1 \ldots j]$.

We can also preprocess the pattern to find matches of prefixes of the pattern within the pattern itself. The **failure array** F of size m is defined as follows: $F[j]$ is defined as the length of the largest prefix of $P[0 \ldots j]$ that is also a suffix f $P[1 \ldots j]$. Note that $F[0] = 0$. If a mismatch occurs at $P[j] \neq T[i]$, we set $j = F[j-1]$.

We can describe this algorithm as:

```
failureArray(P[0..m-1]):
  F[0] = 0
  i = 1
  j = 0
  while i < m:
    if P[i] == P[j]:
      F[i] = j + 1
      i = i + 1
      j = j + 1
    else if j > 0:
      j = F[j-1]
    else:
      F[i] = 0
      i = i + 1
  return F

KMP(T[0..n-1], P[0..m-1]):
  F = failureArray(P)
  i = 0
  j = 0
  while i < n:
    if T[i] == P[j]:
      if j = m - 1:
        return i-j
      else:
        i = i + 1
        j = j + 1
    else:
      if j > 0:
        j = F[j-1]
      else:
        i = i + 1
  return FAIL
```

The `failureArray` function has a runtime of $\Theta(m)$ and the `KMP` function has a runtime of $\Theta(n)$. So we have overall runtime $\Theta(n + m)$ for the KMP method.

6.3 Boyer-Moore

The **Boyer-Moore** algorithm is based on three key ideas:

- Reverse-Order Searching: we compare P with a subsequence of T moving backwards
- Bad Character Jumps: when a mismatch occurs at $T[i] = c$:
  - if $c \in P$, we shift P to align the last occurence of c in P with $T[i]$
  - otherwise, we shift P to align $P[0]$ with $T[i+1]$
- Good Suffix Jumps: if we have already matched a suffix of P before getting a mistmatch, we shift P forward to align with the previous occurence of that suffix (with a mismatch from the actual suffix). This is similar to the failure array in KMP.

These key ideas allow us to skip large portions of T.

The **Last-Occurence Function** is a preprocess of the pattern P. $L(c)$ is defined as the largest index i such that $P[i] = c$ (or -1 if that index does not exist). This can be computed in $O(m+|\Sigma|)$, where Σ is the alphabet. In practice, L is stored in a Σ-sized array.

The **Suffix-Skip Array** is similar: again, we preprocess the pattern P. The array S of size m is definedas: $S[i]$ is the largest index j such that$P[i+1 \dots m-1] = P[j+1 \dots j+m-i-1]$ and $P[i] \neq P[j]$. In this calculation, any negative indices are considered to make the given condition true (these correspon to letter which we may not have checked yet). Thus the SSA is similar to the KMP failure array with an extra condition. It is also computed in $\Theta(m)$.

This algorithm can be represented as:

```
boyerMoore(T[0..n-1], P[0..m-1]):
  L = lastOccurenceArray(P)
  S = suffixArray(P)
  i = m - 1
  j = m - 1
  while i < n and j >= 0:
    if T[i] = P[j]:
      i = i - 1
      j = j - 1
    else:
      i = i + m - 1 - min(L[T[i]], S[j])
      j = m - 1
  if j == -1:
    return i + 1
  return FAIL
```

The worst case running time of this algorithm is $O(n + |\Sigma|)$, though this is difficualt to prove. In practice, the worst case is incredibly rare for this algorithm; this algorithm is almost always faster than KMP for searching English text.

6.4 Tries

A **trie (radix tree)** is a dictionary for binary strings: it is a binary tree based on bitwise comparison and is, in practice, similar to radix sort. Items are stored only in leaf nodes, a left child corresponds to a 0 bit, and a right child corresponds to a 1 bit. Keys can have varying numbers of bits. A trie is **prefix-free**: no key is a prefix of another key.

To search in a trie, we simply follow the correct node until we either end at a leaf (success) or a child does not exist (failure).

To insert into a trie, we perform the following steps:

- search for x
- if we end at a leaf with key x, x must already be in our trie: quit
- otherwise, if we end at a leaf $y \neq x$: this is not possible by our prefix restriction
- otherwise, if we finish at an internal node and have no remaining bits: this is not possible by our prefix restriction
- otherwise, we expand the trie by adding necessary nodes corresponding to remaining bits

To delete from a trie, we simply

- search for x (to find leaf v_x)
- delte v_x and all of its ancestors untilwe reach an acestor with two children

The time complexity of each of these operations is $\Theta(|x|)$ where $|x|$ is the length of the binary string.

6.4.1 Compressed (Patricia Tries)

Patricia Tries reduce the storage requirement by eleminating nodes with only one child. They compress paths into single edges; in fact, a Patricia trie storing n items always has $n - 1$ edges.

Each non-leaf node in a Patricia trie stores an index containg the index of the net bit to-be-tested.

Searching follows the edges down to a leaf. If we end on a node, we are unsuccessful; if we end on a leaf we must check again to see if the key stored in this leaf is x.

To insert, we first search. If the search ends at a leaf L with key y, we compare x against y to determine the first index i where they disagree. We create a new node N with this index I and insert N along the path from the root to L so that the parent of N has index $j < i$ and one child of N is either L or an existing node on the path from the root to L that has index $k > i$. The other child of N will be a new leaf containing x. Otherwise, if the search ends at an internal node, we find the key corresponding to that internal node and proceed similarly to the previous case.

Deleting simply deletes the leaf and its parent.

6.4.2 Multiway Tries

To represent strings over some fixed alphabet Σ, we may use **Multiway Tries**. Each node has at most $|\Sigma|$ children.

Multiway tries do allow strings which are prefixes of other strings: we append an end-of-word character to each item so as to ensure this is possible.

We can also compress multiway tries much in the same way as standard tries.

6.4.3 Suffix Tries

If we want to search for many patterns $P[0 \dots n]$ within the same fixed text T, we find it easier to preprocess T than P. We use the idea that "P is a substring of T if and only if P is a prefix of some suffix of T".

We build a compressed trie which stores all suffixes of T, then insert these suffixes in decreasing order of length. If a suffix is a prefix of another suffix, we do not insert it. We store two indices l and r on each node v (both internal nodes and leaves) where node v corresponds to substring $T[l \dots r]$.

To search for pattern P of length m, we simply:

- search the compressed trie normally (look for a prefix match rather than an entire match)
- if we reach a leaf with a corresponding string length less than m, the search was unsuccessful
- otherwise, we reach a node v (leaf or internal) with a corresponding string length of at least m
- we then only need to check the first m characters of that string to see if there is indeed a match

6.5 Overall Complexities

Overall, then we have the following complexities:

Algorithm	Pre-processing	Search Time	Extra Space
Brute Force	-	$O(nm)$	-
KMP	$O(m)$	$O(n)$	$O(m)$
Boyer-Moore	$O(m + \|\Sigma\|)$	$< O(n)$	$O(m + \|\Sigma\|)$
Suffix Tries	$O(n^2)$	$O(m)$	$O(n)$

7 Compression

We can measure the quality of encoding (compression) schemes by processing speed, security, size, reliability... when determining the quality of a compression algorithm, specifically, we mostly

focus on the size.

Encoding schemes which attempt to minimize the size of the compressed code $|C|$ perform **data compression**. We measure their **compression ratio** as

$$\frac{|C| \log |\sum_{c \in C}|}{|S| \log |\sum_{s \in S}|}$$

There exists both **logical** (based on the meaning of the data) and **physical** (based on the data bits) compression schemes. Furthermore, there exists both **lossless** (recover S exactly) and **lossy** (more size-efficient, but only returns an approximation) algorithms.

Character encodings such as ASCII are encoding algorithms. ASCII, specifically, maps seven bits to encode 128 characters; to decode this, we simply look up the seven-digit code in a **decoding decoding**. The decoding dictionary D:

- must be prefix-free,
- might be used and stored explicitly (e.g. as a trie), or only implicitly, and
- might be agreed in advance (fixed), stored alongside the message (static), or stored implicitly within the message (adaptive)

ASCII is a **fixed-length code** since each key in D has length seven.

7.1 Variable-Length Codes

If different key strings in D have different lengths, we have a **variable-length code**. The UTF coding scheme uses this type of formatting; every unicode character is represented with between one- and four-bytes. ASCII characters are signified by a zero followed by the seven-digit ASCII code, all other characters are represented by between one and four one's (signifying the total number of bytes) followed by a zero and three to five bits. The remaining bits are eight digits beginning with 10.

7.2 Run-Length Encoding

A **run-length encoding** is a variable-length encoding within a fixed decoding dictionary which is not explicitely stored. The source and coded alphabets are both binary: since this will clearly give us runs of zero's and one's, we can encode this as the first bit of S followed by a sequence of integers representing run lengths.

The encoding of run-length k must be prefix-free so the decoder knows when to stop reading. We thus encode the length of k in unary followed by the actual value of k in binary.

The binary length of k is $\text{len}(k) = \lfloor \log k \rfloor + 1$. Since $k \geq 1$, we will encode $\text{len}(k)1$, which is at least 0. The prefix-free encoding of the positive integer k is in two parts: $\lfloor \log k \rfloor$ copies of 0 followed by the binary representation of k.

The compression ratio, then, can be as low as below 1%. Unfortunately, this is rarely the case: there is no compression until $k \geq 6$ and we actually have expansion if $k = 2$ or 4.

7.3 Huffman Coding

We note that certain source-code items are bound to appear much more frequently than others (e.g. when encoding English, the letter e will appear much more frequently than any other letter). We can find it useful, then, to use shorter codes for more frequent letters.

When the source alphabet is arbitrary and the coded alphabet is binary, we can build a binary trie to store the decoding dictionary D. Each character in the alphabet is a leaf in this trie.

We can build the best tree with by:

1. determining the frequency of each character $c \in S$
2. making a height-zero trie for each character (also assign a "weight" to each trie equal to the sum of the frequencies of its contents)
3. recursively merge two tries with the least weights and update the new trie's weight until you only have a single trie D

Note than a min-heap is best for this process.

A Huffman-encoded set of compressed text requires the encoder to do lots of work: it must build the decoding trie, construct the encoding dictionary, and encode the text. The decoding trie must be transmitted along with the encoded text, but this does significantly improve decoding speed.

7.4 Adaptive Dictionaries

Huffman coding uses a **static** dictionary, instead of the **fixed** dictionaries of previous examples: the dictionary is the the same for the entire encoding and decoding processes.

An **adaptive** dictionary has the following properties:

- there is a (usually fixed) initial dictionary D_0
- for $i \geq 0$, D_i is used to determine the ith output character
- after writing the ith character to output, both encoder and decoder update D_i to D_{i+1}

Usually adaptive encoding and decoding algorithms have identical costs.

We can use the **move-to-front** heuristic to take advantage of data locality; any MTF-based compression algorithm will be adaptive.

We can explain MTF encoding as:

```
MTF-encode(S):
  while S.characters:
    c = S.pop_next_character
    print L.find(c)
    L[0], L[i] = L[i], L[0]

MTF-decode(C):
  while C.characters:
```

```
i C.pop_next_integer
print L[i]
L[0], L[i] = L[i], L[0]
```

We can also find certain substrings which appear more frequently (i.e. not just single characters. For English, we have bigrams and trigrams: *th*, *er*, *on*, ..., *the*, *and*, *tha*, ...)

7.4.1 Lempel-Ziv

Lempel-Ziv algorithms are a family of adaptive compression algorithms which encode both characters or substrings.

They tend to have fixed-width encoding using k bits to store the decoding dictionary with q^k entries. The first n elements are for single characters, the remaining are for multiple character subtrings.

After encoding or decoding a substring $y \in S$, we add xc to D where x is the previously encoded or decoded substring of S and c is the first character of y.

7.4.2 Burrows-Wheeler Transform