Part A

1. A Markov chain is a discrete-time stochastic process, defined by a set of states, a set of transition probabilities (between states), and a set of initial state probabilities; the process proceeds probabilistically from state to state according to the time steps, depending only on the previous state (and transition probability). Markov chains are used in various “link analysis” algorithms for IR, like PageRank, and in HMMs, which are often used to POS tagging, chunk tagging, etc. in NLP.

2. An information need is a conceptual description of the problem that the user is trying to solve with an IR engine, approximated by a query. For example, a user might issue the query University of Melbourne when desiring to go to the physical location corresponding to the institution — in which case, they probably want a map — or perhaps the user would like to navigate to the homepage of the institution on the World Wide Web — in which case, they simply want a URL.

3. One approach to compression in IR is compressing the d-gaps (gaps between document identifiers) in a postings list using variable byte compression. Since we need to look at the leading bit of each value, along with the data bits, the time complexity of decompression is $O(n \log n)$ (where n is the total number of postings in the collection; as compared to raw integers which is $O(n)$). This method doesn’t require any extra space when decompressing; assuming that many of the values are small (because the d-gaps for frequent terms are less than 127 (7 bits)), we might expect a space compression of about 75% (8 bits compared to 32-bit ints).

4. Recall usually isn’t important in IR evaluation (because there are far too many potentially relevant documents for a given user to realistically look at); however, in some disciplines like Medicine or Law, a user will typically want to access every single relevant document. (Any single item could be of critical importance; the collection is typically also small compared to, say, the World Wide Web.) It also plays a role in numerous desirable evaluation metrics, like MAP. To assess recall, we would need to examine every single document in the collection for relevance, which would be extremely time-consuming (and exhausting!) for a human adjudicator.

5. Okapi BM25 includes a component for document length (as compared to the average document length in the collection), which is used to down-weight the term frequencies of long documents, and up-weight short documents. It also contains a term for term frequency within the query, so that repeated words in the query receive more importance in the overall calculation.

6. The n-gram method of language modelling assumes that we can evaluate the joint probability of a sequence $P(w_1w_2\cdots w_M)$, according to just $n-1$ words of context in the chain rule expansion. Namely (for the 2-gram case; other cases are similar):

$$P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1w_2)\cdots P(w_M \mid w_1w_2\cdots w_{M-1}) \equiv P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2)\cdots P(w_M \mid w_{M-1})$$

Of course, it is often the case that other factors outside the $(n-1)$-word context window (further back, or even forward) would alter the probability distribution for an individual token. For example, so-called garden path sentences like The horse walked past the barn lay down.

7. The recursive property of languages motivates tree structures. For example, we might observe that a noun phrase can be modified by a prepositional phrase, which contains a noun phrase that contains another prepositional phrase, and so on: the cat on the table in the house on the outskirts of the village by the river through the valley; hence we build rules like NP--$\rightarrow$Det N PP; PP--$\rightarrow$P NP.

8. A hypernym is a specific case of a more general word, for example *cat* is a hypernym of *animal.*
Part B Phrasal IR

1. When constructing a positional index, we build the inverted index as usual, but we alter the postings lists to also include the index(ices) of where the term appears in the document; i.e. rather than:

   \[ t; f_t \rightarrow \{ \langle d : f_{d,t} \rangle \} \]

   we use:

   \[ t; f_t \rightarrow \{ \langle d : f_{d,t}; p_1, p_2, \ldots, p_{f_{d,t}} \rangle \} \]

   When querying, we start with the least frequent term (lowest \( f_t \)) in the query, and keep track of the documents where it has occurred, along with the indices where it has occurred in those documents. Then we proceed with the other terms of the query, rarest to most common, checking that they are present in each of these documents with a corresponding index offset. For example, if the terms are two apart in the query, then the second term must have an index in the document two greater than an index of the first word. Document indices that don’t correspond to both terms can be rejected; documents where we have rejected all positions of one of the query terms can be rejected. The result set is all un-rejected documents when all of the query terms have been processed.

2. Both the space and time complexity of the positional index is \( O(\mid w \mid) \), for the number of words in the collection \( \mid w \mid \). This occurs because, for each word in the collection, we need to store the position where it occurs in the document in some postings list.

3. Compared to a bi-word index, a positional index is much larger, which is a disadvantage if we wish to store it in main memory. On the other hand, we typically would need to post-process the results of a query on a bi-word index, to prune spurious results, which is time-consuming — using a positional index, we do not need to do this.

Link analysis

1. A random surfer can start at any page (hence the uniform initial distribution \( \pi_0 \)). From that page, they can either click a link (with probability \( 1 - \alpha \)) or “randomly teleport” to a new page (by entering a URL into the address bar, with probability \( \alpha \)).

   The transition probability from page \( i \) to page \( j \) corresponds to either: (a) arriving at \( j \) via teleport, with probability mass \( \alpha \times \frac{1}{N} \) (as it is equally likely that we teleport to any page), or (b) following a link from \( i \) to \( j \), with probability mass \( (1 - \alpha) \times \frac{l_{ij}}{l_i} \) (where \( l_{ij} = 1 \) if there is a link to \( j \) on page \( i \), and 0 otherwise; \( l_i \) is the total number of out-links on page \( i \), of which we choose one randomly).

   The sum of these two components makes up each entry of a transition matrix \( P \), with a row and a column for each page. By (pre-)multiplying \( P \) with the column vector according to the initial distribution, we have an iterative method which will (hopefully) converge to the PageRank weights for each page.

2. Our iterative update equation is:

   \[ \pi_n = P \times \pi_{n-1} \]

   which describes the transitions to the next state \( \pi_n \) given the previous state occupancy, \( \pi_{n-1} \).

   **Shorter answer** We are interested in state occupancy \( \pi_n \) in the limit as \( n \rightarrow \infty \), which should reach a steady state. I.e., it does not change distribution after a transition. This can be expressed as

   \[ \pi = P \times \pi \]

   This corresponds to a classic Eigenvalue problem, of the kind \( \vec{x} = \alpha A \vec{x} \), which can be solved efficiently giving several solutions for \((\alpha, \vec{x})\), which are eigenvalue, eigenvector pairs. As we are interested in when
the equality holds exactly, we take the principle eigenvector, with eigenvalue $\alpha = 1$. This gives us a solution for $\pi$ (up to a constant).

**Longer answer** We use an eigenvalue method to diagonalise $P$ in terms of an eigenvalue matrix $A$ and a diagonal matrix $D$: $P = ADA^{-1}$. So,

\[
\begin{align*}
\pi_n &= P \pi_{n-1} \\
\pi_n &= ADA^{-1} \pi_{n-1} \\
\pi_n &= ADA^{-1}ADA^{-1} \pi_{n-2} \\
\pi_n &= ADDA^{-1} \pi_{n-2} \\
\pi_n &= AD2A^{-1} \pi_{n-2} \\
\pi_n &= AD^3A^{-1} \pi_{n-3} \\
&\vdots \\
\pi_n &= AD^nA^{-1} \pi_0
\end{align*}
\]

For complicated reasons, all of the entries of $D$ are between -1 and 1. As we’re looking for the steady state, we can then sensibly evaluate $\lim_{n \to \infty} D^n$. We can then evaluate $\pi \equiv \pi_\infty$ as the set of PageRank page weights.

3. We could multiply the calculated TF-IDF weights by the calculated PageRank weights, so that good PageRank documents are ranked higher in the result set.

**Markov Models**

1. A markov model is defined by a set of states, a set of transition probabilities (between states), and a set of initial state probabilities. For a (word-based) $n$-gram Markov model, each state could correspond to a word sequence of length $n - 1$, and the transition probabilities are based around the word sequence (for a word, as compared to the previous $n - 1$ words).

   In a Hidden Markov Model, there is an extra “hidden” layer, because some property of the words cannot be directly observed from the word sequence. For example, the part-of-speech tag of the word. In this case, the states are the (hidden) POS tags, the transition probabilities are based around sequences of POS tags, and each POS tag has an emission probability of observing each word, given this POS tag.

2. **Wordy answer:** The Viterbi algorithm for HMMs is a dynamic programming algorithm for determining the most-likely state sequence for a set of observations. For example, the best POS tags for a sequence of words.

   Let’s consider the sentence `bear house pest`; we have some set of states, that we will assume consists solely of $J,N,V$ for simplicity.

   The Viterbi algorithm works by progressively estimating the best joint distribution for the first words of the sentence together with the corresponding tag sequence. In this case, we will begin by estimating the probability of beginning with $J$ for `bear`, and $N$, and $V$. We can do this simply by multiplying the initial state probabilities by the corresponding emission probability for `bear`.

   We will use these probabilities, along with the transition probabilities from $J$ to $J$, $N$ to $J$, and $V$ to $J$ to estimate the best sequence where `house` is $J$; similarly for $N$ and $V$ for `house`. We can then multiply the probability of the best sequence with the emission probability of `house`, for each of the three states.

   These will then be the inputs for the `pest` entries, which will proceed the same way. Whichever probability is highest at this point (combining the best transition from the previous probabilities with the emission probability of `pest`) will correspond to the best sequence. (And we’ve kept back-pointers to record what this sequence was! Did I mention that? :) )

   Since, for each word in the sentence, we consider each tag, and the probability that it was immediately preceded by every tag, the complexity is $O(|w| \cdot |t|^2)$ (the number of words in the sentence, times the
square of the number of states).
(It would be worth also illustrating the sentence as a picture, showing the lattice of states.)

Mathematical answer: The Viterbi algorithm searches for the best path (sequence of states) in a HMM given a sequence of observations. This is framed as

$$\arg\max_{s_1, s_2, \ldots, s_T} p(x_1, x_2, \ldots, x_T, s_1, s_2, \ldots, s_T)$$

where $s_i$ is the $i^{th}$ state and $x_i$ is the $i^{th}$ observation. Using the formulation of the HMM, this can be expressed more simply as

$$\max_{s_1, s_2, \ldots, s_T} p(s_1)p(x_1|s_1)p(s_2|s_1)p(x_2|s_2) \cdots p(s_T|s_{T-1})p(x_T|s_T)$$

$$= \max_{s_1} p(s_1)p(x_1|s_1) \max_{s_2} p(s_2|s_1)p(x_2|s_2) \cdots \max_{s_T} p(s_T|s_{T-1})p(x_T|s_T)$$

where we have swapped to max for simplicity (but see back-pointer discussion below). The Viterbi algorithm uses dynamic programming to represent each component in Equation 2, i.e.,

$$\alpha[i, s_i] = p(s_i)p(x_i|s_i)$$

$$\alpha[2, s_2] = \max_{s_1} p(s_2|s_1)p(x_2|s_2)$$

$$\cdots$$

$$\alpha[T, s_T] = \max_{s_{T-1}} p(s_{T-1}|s_T)p(x_T|s_T)$$

such that Equation 2 can be expressed as

$$\max_{s_T} \alpha[T, s_T]$$

And the argmax can be recovered by storing back-pointers which record the ‘winning’ previous state for each cell of $\alpha[i, s_i]$. There are $T \times S$ values in the $\alpha$ dynamic programming matrix (space complexity $O(TS)$), and the computation of each cell involves enumerating $S$ prior states, so the time complexity is $O(TS^2)$. Then give an example sentence / picture with a quick sentence explanation on how this relates to the maths above.

Machine translation

1. Word alignment is the process of automatically identifying how the words of a sentence in the source language map to the words of the translated sentence in the target language, for a given parallel (sentence-aligned) corpus.

   It is often a necessary step in learning a machine translation system, because it allows us to ascertain how words are translated between the two languages without the aid of a complete translation dictionary. This is necessary if we wish to translate novel data, although perhaps memory-based translation is adequate if we have a large enough data set.

2. Phrase–based translation treats word n-grams as the translation units; word–based translation is phrase–based translation for the special case where $n = 1$.

   Taking larger fragments of the sentence to translate allows us to find common translation fragments (like multi-word expressions or idioms, which are usually not well-handled in word–based systems), as well as common re-ordering patterns (like different pronoun or negative-particle orders in some languages). Avoiding the need for many decisions about reordering and often ambiguous word translation improves the system predictions. Larger fragments also have the benefit of being internally coherent, leading to more coherent overall output.

Part C
1. First, we need a representation of the query. We’re told not to use IDF; for a single query term, IDF is irrelevant anyway. Assuming a VSM for \{soccer, football, pitch, hockey, tournament\}, our query is \langle 1, 0, 0, 0, 0 \rangle.

We will also need the document lengths. These are easy here:

\[
\begin{align*}
|d_1| &= \sqrt{3^2 + 0^2 + 4^2 + 0^2 + 0^2} \\
&= \sqrt{9 + 16} = \sqrt{25} = 5 \\
|d_2| &= \sqrt{0^2 + 6^2 + 8^2 + 0^2 + 0^2} \\
&= \sqrt{36 + 64} = \sqrt{100} = 10 \\
|d_3| &= \sqrt{1^2 + 0^2 + 0^2 + 2^2 + 2^2} \\
&= \sqrt{1 + 4 + 4} = \sqrt{9} = 3
\end{align*}
\]

Now we can apply our cosine similarity model; we’re going to calculate the dot product of the query vector with the given document vectors (\langle 3, 0, 4, 0, 0 \rangle for \langle d_1 \rangle etc.) — but in this case, that is just going to be the value of the soccer dimension. We’re then going to divide by the document length (the query length is 1):

\[
\begin{align*}
\cos(q, d_1) &= \frac{q \cdot d_1}{|q||d_1|} \\
&= \frac{3}{(1)(5)} = 0.6 \\
\cos(q, d_2) &= \frac{0}{10} = 0 \\
\cos(q, d_3) &= \frac{1}{3} \approx 0.33
\end{align*}
\]

The ranked order of the documents, from higher assumed relevance to lowest, is 1, 3, 2.

2. The query representation is as above \langle 1, 0, 0, 0, 0 \rangle. Assuming that the top-ranked document is relevant, \langle d_1 \rangle is in our \(D_r\) set. Since \(\gamma = 0\), we can ignore the \(D_{nr}\) set.

Just before we substitute, we want the document vector to be of unit length, so we divide through by the length that we calculated above: \(d_1 = \langle 0.6, 0, 0.8, 0, 0 \rangle\). Now:

\[
q_e = \alpha q + \beta \frac{1}{d_1} d_1 + 0 \cdots
\]

\[
= (0.5)(1, 0, 0, 0, 0) + (0.5)(0.6, 0, 0.8, 0, 0)
\]

\[
= (0.8, 0, 0.4, 0, 0)
\]