Project 1

# Simple Search Engine

系所：資工所

姓名：李晟維

學號：P76044295

# 1. Introduction

這是實作一個小搜尋引擎的作業，學習如何將三萬多筆的網頁資料進行切割，並有效率的建立索引，讓使用者可以在進行查詢時，大幅提高速度，並出現想要得到的結果。

# 2. Environment

作業系統 : win10　64bit

開發語言 : java

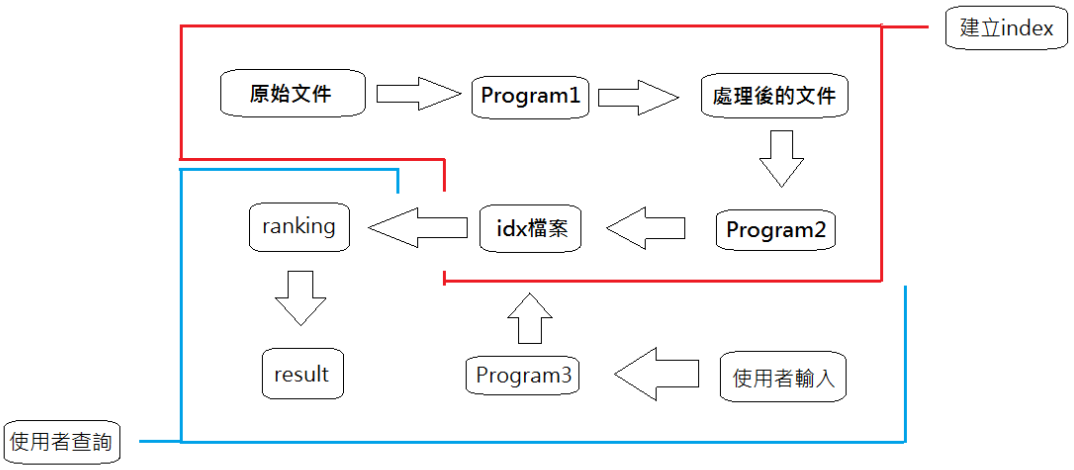
開發環境 : eclipse

Cpu : E3 – 1231 V3

Memory : 16G

Disk : SSD M6S 128G / HDD SEAGATE 1TB

# 3. System Architecture




(a) 資料處理部分

首先原始文件會先經過程式 Program1 處理，將中文英文進行分別處理後寫成處理後的文件。

EX : (0.txt ~31042.txt)

而處理後的文件經過程式 Program2 的處理後，會建立 index，分成 4096 個檔案。

EX : (0_idx.txt~4096_idx.txt)

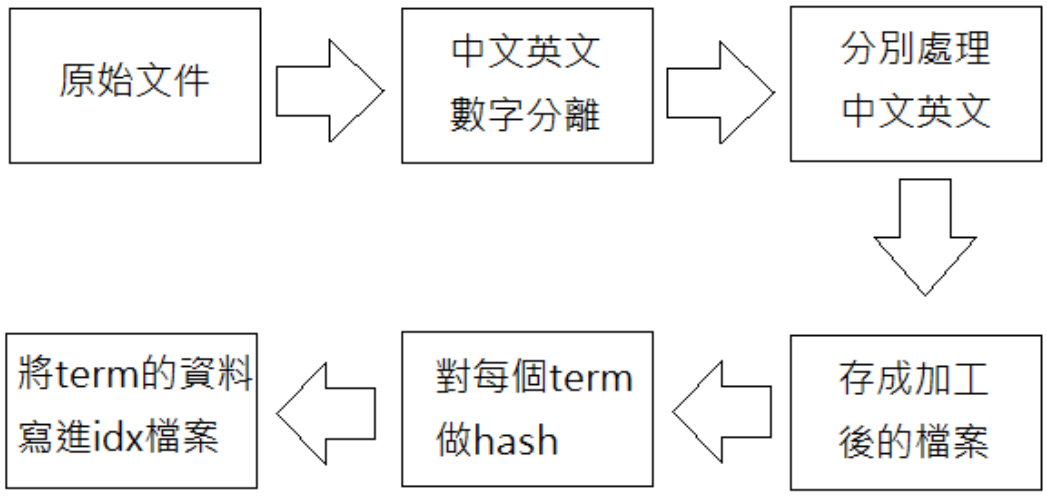
(b) 使用者查詢部分

使用者輸入的 query 同樣經過 Program3 來進行字串處理，並與 index 檔作比對，之後算分數，

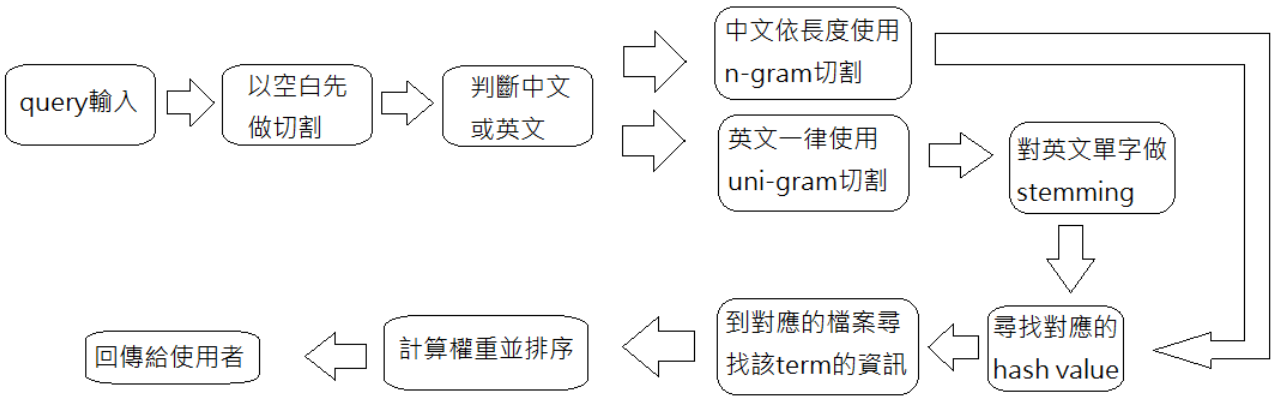
把結果回傳給使用者。

# 4. System Flow

(a) 處理 data，建立 index




(b) 處理使用者查詢



# 5. Detail Technology

(a) Normalization

首先會將文章的標點符號如逗號，問號等等都去除掉以空白代替，並將剩餘的 term 分為三部分，分別是中文英文與數字。

(1) 中文部分:

起初有兩個想法，一種是使用現成的切字系統，而另一種是自行進行 n-gram 的切割。前者又分為線上與線下，而線上顯然不切實際，除了網路因素外，還需要考慮到伺服器的負載量，而我實驗過的網站都無法支援承受大量資料，所以在選擇上我採用了線下的切字系統 mmseg4j，而這與自行進行 n-gram 切割各有優缺點，優點部分為快速且資料量小。例如我想切割：「我買一顆蘋果」，用定義好的字典檔會切出「我」「買」「一顆」「蘋果」，而使用 n-gram 則必須看你 n 的定義，若以此例來看 n 最高到 6，則會切出 21 個 term，在切

字部分就會花費不少時間，之後建 index 所費的時間更是可觀；而缺點部分顯而易見，對於一些沒有定義過的單字或專有名詞，由於無法判斷，只會切成一個一個字，之後再查詢的時候，無法有效找到想要查詢的 term，之後實驗部分會再舉例說明。

(2) 英文部分:

英文部分一律轉成小寫，並只會進行 uni-gram 切割，之後進行 stemming 處理。

(3) 數字部分:

數字部分不做另外的處理，僅以空白切割。

Ex：「2015/10/19」，會切成「2015」「10」「19」3 個 term。

(b) Stemming

對於相似的英文單字如「worked」「worker」「works」等等，會統一處理成「work」，而使用的方法為 Porter Stemming Algorithm，只是此方法在少數情況下無法準確切出所要的單字，例如 ate -> at ，play ->plai 等等。

(C) N-gram

在處理文件時，由於考量到切割速度與後續建立 index 的時間因素，故沒有採用。但在之後使用者查詢時，若是中文詞，會根據長度進行 N-gram 切割，若長度為 4，就是 4-gram，例如：「中華民國」會切成「中」「華」「民」「國」「中華」「華民」「民國」「中華民」「華民國」「中華民國」，其中根據字的長度會給予額外的倍率 n^2 ，例如如果 match 到「民國」，分數會額外乘上 2^2=4 倍，以此類推，如此的用意是為了確保單字長度越高所獲得的鑑別度越好。

(D) TF & TF-IDF

(a) TF：詞頻，公式如下:

$tf_{i,j} = \frac{freq_{i,j}}{freq_{max,j}}$ ，某個 term 在同一篇文章中出現的次數，除以整個文章的 term 個數，若分數越高，則表示這個 term 在該文章中越重要，這部分在建立 index 時，就已經處理完畢存在每個 term 的後面了。

(b) TF-IDF：詞頻-逆向文件頻率，公式如下:

$idf_{i,j} = log\frac{N}{n_i}$ $w_{i,j} = tf_{i,j} * idf_{i,j}$ ，除了剛剛的 TF

外，而多增加了一個考量因素 IDF，其值為全部文章數量除以出現某個 term 的文章數量再取 log，用意是若某個單字出現的次數越少，則越有鑑別度，所得到的 IDF 值越高，而這部分是在 search 的時候進行計算的。

# 6. Performance Discussion

(a) 處理原始 31042 個檔案檔案，進行字串切割與 stemming，分別寫成 0 到 31042 個 txt 檔，由於是使用現成的字典檔做切割，並且在 SSD 上面讀寫，故整個過程約 5 分鐘，不過後來擔心大量讀寫會導致 SSD 壽命縮短，故後面的動作都移至 HDD 上。

(b) 將 term 寫入 4096 個 index 檔案，我個人的做法是每次處理 1000 筆 txt 檔，先把資訊 load 到 memory 上做處理，再寫進 index 檔案，寫完後再繼續處理下 1000 筆檔案，直到完成 term 的 index，整個過程約耗時 25 分鐘。

(c) 使用者查詢，英文的話由於是 uni-gram，查詢速度非常快，約 1 秒左右，而中文的話，則是隨著 term 的長度做變化，不過基本都是在 3-10 秒內。

(d) Top-k precision v.s k

以下對 6 個 query 進行測試：

(1)化學反應

| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 87.5% | 87.5% | 80% |
| TF-IDF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 72.7% | 66.6% | 61.5% | 57.1% | 53.3% | 56.2% | 52.9% | 50% | 47.3% | 45% |
| TF-IDF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |

化學反應的結果，在 TF 部分，命中約 5 成，而沒有命中的也都有包含化學或是反應；而 TF-IDF 高達 100%命中率。

(2)最短路徑演算法

| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 0% | 0% | 0% | 0% | 0% | 16.6% | 14.2% | 12.5% | 11.1% | 10% |
| TF-IDF | 0% | 0% | 0% | 25% | 40% | 33.3% | 28.5% | 25% | 22.2% | 20% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 9% | 8.3% | 7.6% | 7.1% | 6.6% | 6.2% | 5.8% | 5.5% | 5.2% | 5% |
| TF-IDF | 18.1% | 16.6% | 23% | 21.4% | 20% | 18.7% | 17.6% | 16.6% | 15.7% | 15% |

可以看到 TF 與 IDF 的命中率都不是很高，因為我有額外加權的關係，所以找出來的文章幾乎都與演算法有關，很多篇也都是計算最短路徑的，只是沒有抓到完整的這 7 個字。

(3) Gaussian multiple-input multiple-output broadcast channel

| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| TF-IDF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| TF-IDF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

可以看到結果非常悽慘，完全沒命中到，我個人認為原因有 2 個可能。第一：我英文使用 uni-gram 做搜尋，導致搜出來的文章幾乎都是某個特定單字如特別多，例如 channel；而第二種可能，便是我沒有處理「-」符號，原本 multiple-input 的 idf 值應該是會很高的，但被我拆成「multiple」與「input」，從而導致大量文章都有這些字，造成搜尋效率不佳。

(4)高宏宇

| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| TF-IDF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |
| TF-IDF | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

毫無意外，在搜尋的時候就料到是這結果了，而這也是使用現成切字系統的壞處，因為高宏宇為專有名詞，切字系統不知道這三個字，在辭檔裡會被切成「高」「宏」「宇」，而這三個字在某些文件出現率極高! 造成根本搜尋不到的情況。我認為解決方法有兩種，第

一種就是建 index 時改成 n-gram 切割，這樣我相信前幾筆肯定會搜尋到高宏宇，之後可能會參雜些李宏宇、高宏泰之類的(假如原始檔有的話)，而另一種方法就是在現有的辭檔裡加入「高宏宇」，這樣的話我相信 hit 率應該超過 90%，不過這方法有點作弊，我就沒有使用了。

(5)2015

| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| TF-IDF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |
| TF-IDF | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% | 100% |

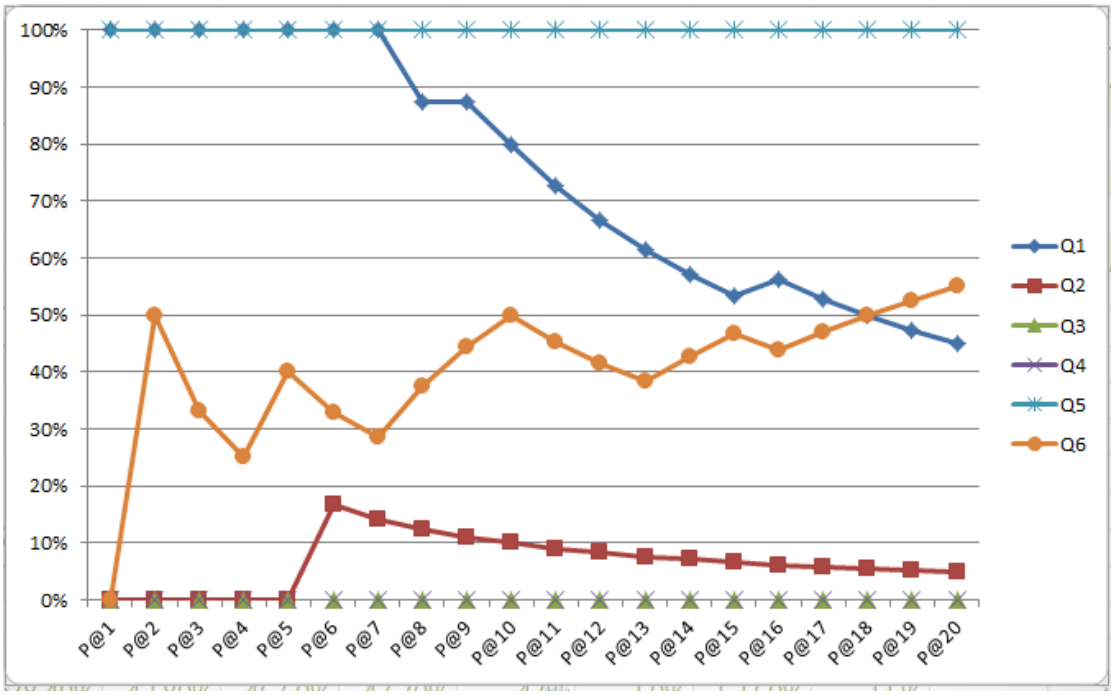
這部分也是在意料之中，畢竟數字我沒有特別進行處理，所以只要原始文件有，就一定會 hit 到，而使用者輸入 2015，所預期得到的應該都是與 2015「年」有關的，也就是時間方面的，而所 match 到的 20 篇，無論是 TF 或是 TF-IDF，都與 2015 年有關。

(6)影像處理

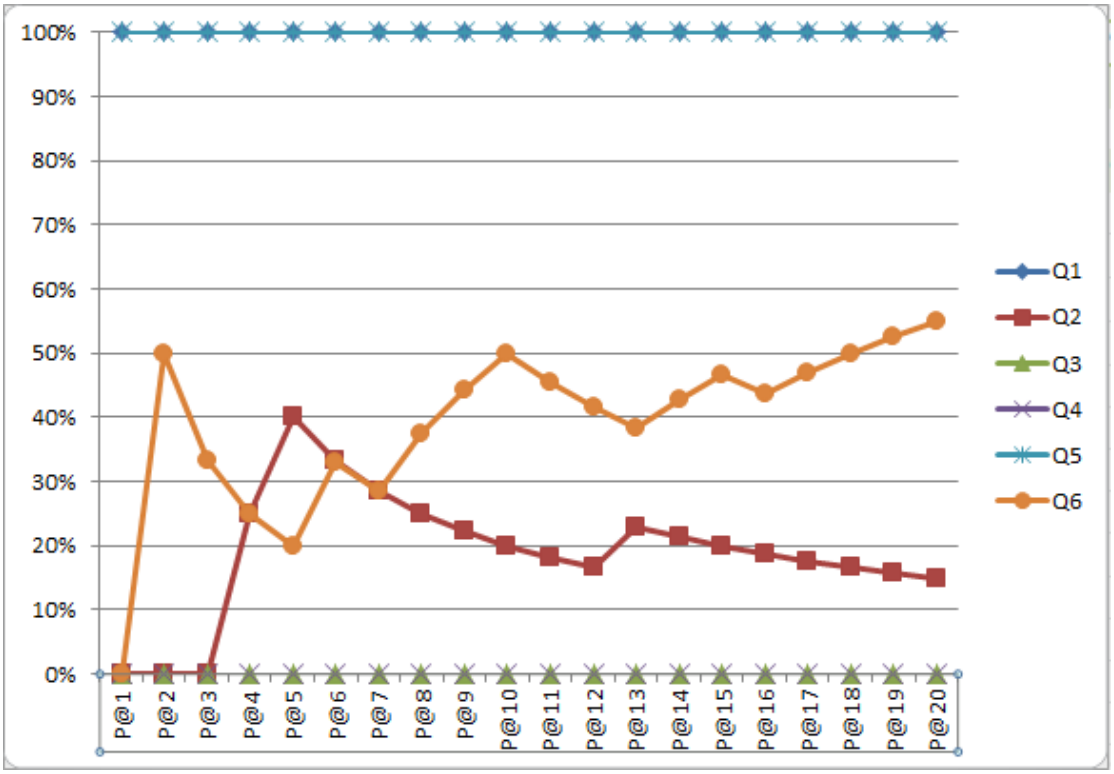
| P@K | P@1 | P@2 | P@3 | P@4 | P@5 | P@6 | P@7 | P@8 | P@9 | P@10 |
|---|---|---|---|---|---|---|---|---|---|---|
| TF | 0% | 50% | 33.3% | 25% | 40% | 33% | 28.5% | 37.5% | 44.4% | 50% |
| TF-IDF | 0% | 50% | 33.3% | 25% | 20% | 33% | 28.5% | 37.5% | 44.4% | 50% |
| P@K | P@11 | P@12 | P@13 | P@14 | P@15 | P@16 | P@17 | P@18 | P@19 | P@20 |
| TF | 45.4% | 41.6% | 38.4% | 42.8% | 46.6% | 43.7% | 47% | 50% | 52.6% | 55% |
| TF-IDF | 45.4% | 41.6% | 38.4% | 42.8% | 46.6% | 43.7% | 47% | 50% | 52.6% | 55% |

可以看到這個查詢的命中率不論是 TF 或 TF-IDF 都有不錯的表現，而這兩者的推薦結果可以發現有許多一樣的地方，和第一個 query「化學反應」不太一樣。我想這是因為「化學」這個詞有其獨特性，從而使 idf 值比較高，因此造成 query1 的 TF 與 TD-IDF 之間造成一段差距。而此例子中，「影像」與「處理」之間數量應該是相差不多，才會造成相近的結果。

## Top-k precision v.s k (TF)




## Top-k precision v.s k (TF-IDF)



# 7. Discussion

這作業有別於我以往所寫過的任何程式作業，行數不多，但寫了非常非常久，遇到了好多好多問題，但也從中學習到了好多東西。

(1) 首先是關於開檔關檔的問題。以前寫程式的時候，從來沒有 close 的習慣，反正讀取的是小檔案，程式跑完後也會自己關掉，一直都沒有注意到這種細節，當檔案一多，問題就出現了，很可能造成重複讀取或記憶體漏失等莫名其妙的 bug。

(2) 如何有效地切割字串建立 index，也是這作業的核心之一，可以看到我所使用的切割方式，雖然前置處理時間很短，但針對某些特定的名詞，如人名效果就非常差，而若是完全使用 n-gram 切割，所費時間必定很長，因此如何取捨是個大問題。我個人有個想法，先利用寫好的辭檔進行切割，若切出來的 term 大於 2 個中文字則保留，然而若連續 2 次以上切出 1 個中文，則進行合併，直到下個切出的 term 大於 2 個中文，然後對其

做 n-gram 切割。例如：「高宏宇努力」，一般切字系統會切出「高」「宏」「宇」「努力」，而這方法會將「高」「宏」「宇」再進行合併，並進行 n-gram。這樣應該感覺可以解決對於特定詞彙命中率過低的問題，所費的時間也不會像單純 n-gram 一樣高。

(3) 記憶體的分配問題。這也是一門學問，以前寫的程式都小小的，從來沒有這問題，有什麼東西就塞進去記憶體就對了。然而這次的作業，卻沒辦法這麼做，如何有效率的控管記憶體，哪時候該釋放掉，都是很重要的，所以我前面有提到，我每讀取 1000 筆資料，放到記憶體理處後過後，寫進檔案後，就會釋放掉，再重新讀取，這也是經過幾次記憶體爆掉的教訓後，才知道的。

(4) 新的語言。其實我本身是對於 C，C++比較熟悉的，但這次我決定學習一個完全陌生的語言來完成這次的作業。其中遇到許多困難，像是以前 C 都習慣自己寫 hash，自己建 array，空間不足就使用指標 linked list 到自己 malloc 的記憶體。但

這些 java 大部分都可以直接幫你處理好，我印象最深刻的就是 java 物件的 reference，如果是 C，我可以透過指標清楚知道目前指向哪個記憶體，然而 java 沒有指標，卻有類似指標概念的 reference，這造成了一個 bug，超過 8 小時以上我都在處理這個 bug，中途換了無數種寫法都錯誤，甚至還一度想用 c 語言來重寫這作業。而最後才知道問題出在哪裡，這很大一方面也是我對 java 不是很熟悉的緣故，不過這也是為什麼我用 java 寫這次作業的原因。

# 8. Question

Q 1: What can you do if your system get nothing?

A1 : 當不滿 20 筆時，有幾筆就輸出幾筆，若是都沒有，則什麼都不輸出。

Q2: What can you do if your system get many results with the same score?

A1 : 在此處我沒有做特別處理，先 insert 進去的在前面，不過應該可以計算兩個文件 match 到有算分的字數，match 到比較少次數地的優先顯示，例如說，當使用者輸入「跑步機」，若文章 1 包含 5 次跑步機，而文章 2 可能只出現 1 次跑步機，卻有一大堆關於「跑步」的資訊，造成分數拉高到與文章 1 一樣，此時文章 1 明顯是我們比較想要的，因此優先輸出。

Q3 : What are the difference between the text processing methods (e.g., indexing, stemming) you applied?

A1 : 在一開始處理原始文件時，就單純去除特殊符號，並以空白做切割而已，而後面建立 index 時中文使用了mmseg 分詞算法，英文使用了 porter algorithm，而在查詢時，中文則使用 n-gram 切割。

Q4 : Can you find different rankings if you applied different weighting schemes?

A4 : 如同前面我說的，在 N-gram 上，我有試著調整 N 的權重，在實作部分我是 n*n，但是當我條成 n*n*n 時，部分 query 的 hit 率會大幅提高。像是 query1 的「化學反應」，在 TF 方法中，hit rate 會從 45%提高到 100% ; 但有些反而會降低，像是在 n*n*n 的情況下，query2 的 TF-IDF 方法中，hit rate 會從 15% 掉到 10%。

Q5 : Can you find different performance ranking if you applied different evaluation metrics ?

A5 : 這部分礙於對 JAVA 不是非常純熟與時間上的關係，我沒有實作其他方法。但從網路上有查到一些相關的算法如 Okapi BM25，感覺應該值得一試。