```
(*有關於LaTeX使用於斷行的演算法

首先我們要制定一個字元（包含斷行後新生的連字號，以及空白）在斷行前、斷行後的寬度

*)
(*假設這段字元 word segment list = wordSegList，SP表達半形空白，HY表達連字號*)
let wordSegList = ["no"; "HY"; "thing"; "SP"; "can"; "SP"; "stop";"SP"; "the"; "SP";
"cro"; "HY"; "co";"HY" ;"dile"; "cross"; "SP"; "it."] ;;
val wordSegList : string list =
  ["no"; "HY"; "thing"; "SP"; "can"; "SP"; "stop"; "SP"; "the"; "SP"; "cro";
   "HY"; "co"; "HY"; "dile"; "cross"; "SP"; "it."]
(*現在轉成每個word segment都附帶長度的格式*)
(* sg: 文段 segment
ow : original width 原來的寬度
hw : hyphenated width 該處指定為斷字後的寬度*)

type segment_with_length = { sg: string; ow: float; hw: float}
type segment_with_length = { sg : string; ow : float; hw : float; }
(* 每個segment之長度*)
let segOwList = List.map (fun x -> match x with
                            | "SP" -> 1.0  (*SP 通常寬度為1*)
                            | "HY" -> 0.0 (*HY 連字點寬度為0*)
                            | _ -> float_of_int (String.length x))    (*以chars的長度來
當做文字寬度 假設是等寬半形字元*)
                                    wordSegList

val segOwList : float list =
  [2.; 0.; 5.; 1.; 3.; 1.; 4.; 1.; 3.; 1.; 3.; 0.; 2.; 0.; 4.; 5.; 1.; 3.]
(*每個segment在其被斷行時的長度 *)
let segHwList = List.map (fun x -> match x with
                            | "SP" -> 0.0  (*SP 通常斷行後寬度為0*)
                            | "HY" -> 1.0 (*HY 連字點斷行後為1*)
                            | _ -> infinity)(*不可能斷行的地方，寬度設做0*)
                                    wordSegList
val segHwList : float list =
  [infinity; 1.; infinity; 0.; infinity; 0.; infinity; 0.; infinity; 0.;
   infinity; 1.; infinity; 1.; infinity; infinity; 0.; infinity]
(*3個列表組合 zip 在一起*)

let segListCombined = List.combine (List.combine wordSegList segOwList) segHwList;;

(*然後變成type segment_with_length的列表*)

let segWithLengthList = List.map (fun i -> match i with
                                    | ((sg,ow),hw) -> {sg = sg; ow = ow; hw = hw})
segListCombined
val segListCombined : ((string * float) * float) list =
  [(("no", 2.), infinity); (("HY", 0.), 1.); (("thing", 5.), infinity);
   (("SP", 1.), 0.); (("can", 3.), infinity); (("SP", 1.), 0.);
   (("stop", 4.), infinity); (("SP", 1.), 0.); (("the", 3.), infinity);
   (("SP", 1.), 0.); (("cro", 3.), infinity); (("HY", 0.), 1.);
   (("co", 2.), infinity); (("HY", 0.), 1.); (("dile", 4.), infinity);
   (("cross", 5.), infinity); (("SP", 1.), 0.); (("it.", 3.), infinity)]
```

```
val segWithLengthList : segment_with_length list =
  [{sg = "no"; ow = 2.; hw = infinity}; {sg = "HY"; ow = 0.; hw = 1.};
   {sg = "thing"; ow = 5.; hw = infinity}; {sg = "SP"; ow = 1.; hw = 0.};
   {sg = "can"; ow = 3.; hw = infinity}; {sg = "SP"; ow = 1.; hw = 0.};
   {sg = "stop"; ow = 4.; hw = infinity}; {sg = "SP"; ow = 1.; hw = 0.};
   {sg = "the"; ow = 3.; hw = infinity}; {sg = "SP"; ow = 1.; hw = 0.};
   {sg = "cro"; ow = 3.; hw = infinity}; {sg = "HY"; ow = 0.; hw = 1.};
   {sg = "co"; ow = 2.; hw = infinity}; {sg = "HY"; ow = 0.; hw = 1.};
   {sg = "dile"; ow = 4.; hw = infinity};
   {sg = "cross"; ow = 5.; hw = infinity}; {sg = "SP"; ow = 1.; hw = 0.};
   {sg = "it."; ow = 3.; hw = infinity}]
(*
我們可以定義在第 n 處斷行=>除了斷行點以外的文字消失，的成本函數 cost(n)，成本函數越小越好。
這時後需要用動態規劃解決。
badness (k, n)是指k~n-1處若塞於一行，且n處斷行時的懲罰函數（等下介紹），越小越好
cost(n) = baness(0,n) 若其為有限，否則 min of k in 0...n-1 of badness(k, n) + cost(k)

懲罰函數badness定義是：若lineWidth >= widthBetween(a,b)，則為二者之差的三次方，否則是無限大。
k >= n
badness(k, n) =  (lineWidth - widthBetween(k, n) )^3 if lineWidth >=
widthBetween(k+1, n)
                infinity                                elsewhere

widthBetween(a,b)係指 a到b 塞在一行時的寬度
widthBetween(a,b) = hw[b] + (sum{i=a...b-1} of ow[i]
*)
open Printf

let lineWidth = 12.0;; (*一行最大寬度*)

let widthBetween a b = if a> b then raise (Failure "Exception: widthBetween a b, a
<=b ")
    else (List.nth segWithLengthList b).hw +. (sumOfOw a (b-1) segWithLengthList);;
let badness k n = let remainedSpaceWidth = lineWidth -. (widthBetween k n) in
                      if remainedSpaceWidth >= 0. then
                          remainedSpaceWidth ** 3.
                          else infinity;;

let minIndex = ref 0;; (*cost(x)發生的最小的k)值*)

(*動態規劃存放 (min cost, 其中的 k 滿足 min cost) 之處*)
(*格式： n (minValue, minIndex) *)
let costKStorage = Hashtbl.create 10;;

let rec cost n =
    if Hashtbl.mem costKStorage n then (*若是已經存儲了，即用裡面的值，避免重複運算*)
        let (minValue, minIndex) = Hashtbl.find costKStorage n in
        minValue
    else if (badness 0 n) < infinity then (badness 0 n)
    else
        let compareList = List.init n (fun k -> (badness k n) +. cost k) in
        (*找最小值*)
        let findMin lst = List.fold_left min infinity lst in
```

```
        let minValue = findMin compareList in (*最小值*)
        (*找最小值所在的索引index值*)
        let findMinIndex lst = List.fold_left
                                    (fun pos i -> if (List.nth lst i) == minValue
then i else pos)
                                    (-1)
                                    (List.init (List.length lst) (fun x -> x)) in
        let minIndex = findMinIndex compareList in
        let _ =  Hashtbl.add costKStorage n (minValue, minIndex) in
        minValue;;
val lineWidth : float = 12.


val widthBetween : int -> int -> float = <fun>


val badness : int -> int -> float = <fun>


val minIndex : int ref = {contents = 0}


val costKStorage : ('_weak11, '_weak12) Hashtbl.t = <abstr>


val cost : int -> float = <fun>
(*sumOfOw : 上文的(sum{i=a...b} of ow[i]*)
(* sumOfOwAux：輔助函數*)
let rec sumOfOwAux i start final sum list =
if i < start then sumOfOwAux (i+1) start final sum list
else if (i>= start && i <= final) then sumOfOwAux (i+1) start final (sum +. (List.nth
list i).ow) list
else sum ;;

let sumOfOw start final list = sumOfOwAux 0 start final 0.0 list;;
val sumOfOwAux :
  int -> int -> int -> float -> segment_with_length list -> float = <fun>


val sumOfOw : int -> int -> segment_with_length list -> float = <fun>
(*算到第11之處的成本*)
(*結果
no thing
can stop
crocodile
...........^
最多只能塞到箭頭處*)
cost 11;;
```

```
- : float = 179.
```

```
(*找 costKStorage 目前的值*)
let a = ref "" in
    let _ = (Hashtbl.iter (fun x y -> let (y1,y2) = y in a := !a ^ (sprintf "%d : %f
%d\n" x y1 y2))  costKStorage) in !a;;
```

```
- : string =
"6 : inf -1\n2 : inf -1\n8 : inf -1\n7 : 152.000000 3\n13 : 153.000000 7\n12 : inf
-1\n4 : inf -1\n9 : 28.000000 5\n11 : 179.000000 7\n0 : inf -1\n10 : inf -1\n"
```

```
(*找出每個斷行點，回溯的搜尋HashTable*)
```

```
let rec findBreakPointAux res k =
if Hashtbl.mem costKStorage k then
    let (minValue, minIndex) = Hashtbl.find costKStorage k in
    findBreakPointAux (List.append res [k]) minIndex
  else (List.append res [k]);;

let findBreakPoint n = findBreakPointAux [] n;;
```

```
val findBreakPointAux : int list -> int -> int list = <fun>
```

```
val findBreakPoint : int -> int list = <fun>
```

```
findBreakPoint 13;;
```

```
findBreakPoint <-- 13
findBreakPointAux <-- []
findBreakPointAux --> <fun>
findBreakPointAux* <-- 13
Hashtbl.find <-- <abstr>
Hashtbl.find --> <fun>
Hashtbl.find* <-- <poly>
Hashtbl.find* --> <poly>
findBreakPointAux <-- [13]
findBreakPointAux --> <fun>
findBreakPointAux* <-- 7
Hashtbl.find <-- <abstr>
Hashtbl.find --> <fun>
Hashtbl.find* <-- <poly>
Hashtbl.find* --> <poly>
findBreakPointAux <-- [13; 7]
findBreakPointAux --> <fun>
findBreakPointAux* <-- 3
findBreakPointAux* --> [13; 7; 3]
findBreakPointAux* --> [13; 7; 3]
findBreakPointAux* --> [13; 7; 3]
findBreakPoint --> [13; 7; 3]
- : int list = [13; 7; 3]
```