```
def longest_common_subsequence(x, y):
    n = len(x)
    m = len(y)
    #                    -- calcul longueur optimale
    A = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(n):
        for j in range(m):
            if x[i] == y[j]:
                A[i + 1][j + 1] = A[i][j] + 1
            else:
                A[i + 1][j + 1] = max(A[i][j + 1], A[i + 1][j])
    #                    -- extraire solution
    sol = []
    i, j = n, m
    while A[i][j] > 0:
        if A[i][j] == A[i - 1][j]:
            i -= 1
        elif A[i][j] == A[i][j - 1]:
            j -= 1
        else:
            i -= 1
            j -= 1
            sol.append(x[i])
    return ''.join(sol[::-1])    # liste inversée
```

Variante avec plusieurs séquences données Supposons qu'on veuille calculer une plus longue sous-séquence commune non pas à 2 mais à k séquences données, de longueur respectivement $n_1, \ldots, n_k$. Alors l'approche ci-dessus se généralise. Il faut calculer une matrice A de dimension k, calculant la plus longue séquence commune pour toute combinaison de préfixes des séquences données. La complexité de cet algorithme est $O(2^k \prod_{i=1}^{k} n_i)$.

Variante avec deux séquences triées Le problème peut se résoudre en temps $O(n + m)$ si les deux séquences sont triées, car dans ce cas on peut procéder comme pour la fusion de deux listes triées (voir section 4.1 page 59).

En pratique L'algorithme BLAST (pour *Basic Local Alignment Search Tool*) est utilisé, mais il ne garantit pas de produire toujours une solution optimale.

3.4 Plus longue sous-séquence croissante

Définition Étant donné une séquence de n entiers x, il faut trouver une sous-séquence de s de longueur maximale et strictement croissante.

Application Considérons une route droite, qui mène à la mer, et des maisons construites le long de la route. Chaque maison est constituée d'un certain nombre d'étages et bénéficie d'une vue sur la mer si toutes les maisons entre elle et la mer ont un nombre d'étages moindre. On aimerait que toutes les maisons aient vue sur la mer et détruire le plus petit nombre de maisons pour arriver à cette fin (voir figure 3.4).

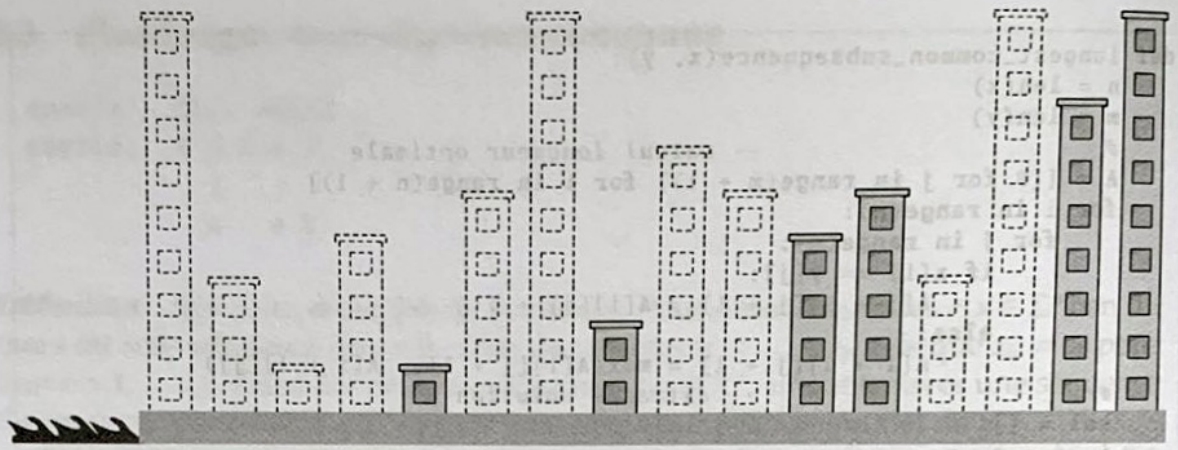

FIGURE 3.4 – Détruire le moins de maisons possibles afin que les maisons préservées aient vue sur la mer.

Algorithme en $O(n \log n)$ Plus précisément, la complexité est $O(n \log m)$ où m est la taille de la solution produite. Dans une approche gloutonne, on veut, pour chaque indice i, essayer de compléter avec x_i une sous-séquence croissante du préfixe $x_1, \ldots, x_{i-1}$. Mais laquelle de ces sous-séquences va aboutir à une solution optimale ? Considérons l'ensemble des sous-séquences croissantes du préfixe. Dans une sous-séquence croissante y deux attributs sont importants : sa longueur, et le dernier élément. L'intuition : on aime bien que les sous-séquences croissantes soient longues — car c'est l'attribut à optimiser — et qui terminent avec un petit élément, puisque ceci permet de les compléter plus facilement.

Pour formaliser cette intuition, notons par $|y|$ la longueur d'une sous-séquence y et par y_{-1} le dernier élément de y. On dit que y *domine* z si $|y| \geq |z|$ et $y_{-1} \leq z_{-1}$ et qu'au moins une des inégalités est stricte. Il suffit de s'intéresser aux sous-séquences non dominées vont pouvoir être complétées en une sous-séquence optimale.

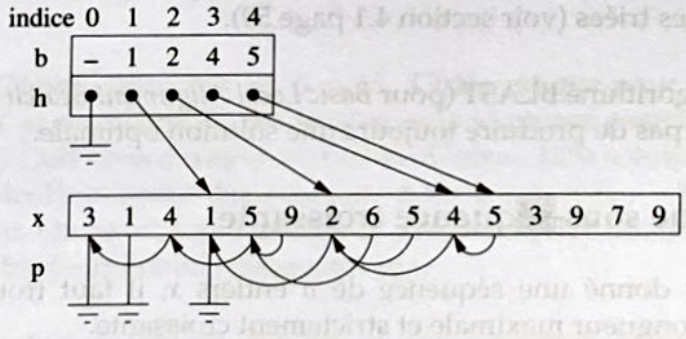

FIGURE 3.5 – Calcul de la plus longue sous-séquence croissante. En gris l'entrée de x en cours de traitement. Pour le préfixe considéré les séquences maintenues sont $(), (1), (1,2), (1,2,4), (1,2,4,5)$. Après le traitement de l'entrée 3, la séquence $(1,2,4)$ sera détrônée par $(1,2,3)$.

Les sous-séquences non dominées du préfixe $x_1, \ldots, x_{i-1}$ diffèrent par leur longueur. Donc pour chaque longueur k on garde une sous-séquence de longueur k terminant par un entier minimal. Concrètement, on maintient un tableau b tel que $b[k]$ soit le dernier élément de la plus longue sous-séquence de longueur k. Par convention on note $b[0] = -\infty$.

L'observation clé est que le tableau b est strictement croissant. Ainsi, au moment du traitement de l'élément x_i, au plus une seule des sous-séquences est à mettre à jour. Notamment si k est tel que $b[k-1] < x_i < b[k]$, alors on peut améliorer la séquence de longueur $k-1$ en la complétant par x_i, et obtenir une meilleure sous-séquence de longueur k, dans le sens qu'elle termine avec un élément plus petit. Dans le cas où x_i est plus grand que tous les éléments de b, on peut augmenter b avec l'élément x_i. C'est la seule amélioration possible, et la recherche de l'indice k peut se faire par dichotomie en temps $O(\log |b|)$, où $|b|$ est la taille de b.

Détails d'implémentation Les sous-séquences sont codées par des listes chaînées à l'aide de tableaux h et p. La tête de la liste est codée par h, tel que $b[k] = x[h[k]]$. Puis le prédécesseur d'un élément j est $p[j]$. Les listes terminent par la constante *None*, voir figure 3.5.

```
from bisect import bisect_left


def longest_increasing_subsequence(x):
    n = len(x)
    p = [None] * n
    h = [None]
    b = [float('-inf')]  # - infinity
    for i in range(n):
        if x[i] > b[-1]:
            p[i] = h[-1]
            h.append(i)
            b.append(x[i])
        else:
            #   -- recherche dichotomique: b[k - 1] < x[i] <= b[k]
            k = bisect_left(b, x[i])
            h[k] = i
            b[k] = x[i]
            p[i] = h[k - 1]
    # extraire solution
    q = h[-1]
    s = []
    while q is not None:
        s.append(x[q])
        q = p[q]
    return s[::-1]
```

Variante sous-séquence non décroissante Si la sous-séquence n'a pas besoin d'être strictement croissante, mais seulement non décroissante, alors au lieu de chercher k tel que $b[k-1] < x[i] \leq b[k]$, il convient de chercher k tel que $b[k-1] \leq x[i] < b[k]$. Ceci se fait à l'aide de la fonction Python *bisect_right*.