トの位置を中心にして，左側には 30 よりも小さな要素，右側には 30 よりも大きな要素がくるようになり，分割が完了した．

問題 6　リスト 3 で示したクイックソートの疑似コードを実際に動作するプログラムにしなさい．具体的には，6〜8 行目に本節で説明した分割アルゴリズムを実装すればよい．

分割アルゴリズムの計算量が $O(n)$ であり，全体では分割アルゴリズムを平均 $\log_2 n$ 回実行することから，クイックソートの平均的な計算量は $O(n \log n)$ であることがわかる．しかし，これはピボットによる分割において常に二等分することができた場合であり，ここで大きな偏りが生じると，最悪の場合には分割アルゴリズムを $n-1$ 回実行しなければならないので，計算量は $O(n^2)$ となる[2]．

問題 7　上で述べたような最悪の場合が起こるのは，分割アルゴリズムのピボット選択において常に右端を選ぶことが原因である．それを避けるために，部分配列の左端，中央，右端の中央値をピボットとして選ぶように分割アルゴリズムを変更しなさい．

4　動的計画法による配列のアラインメント

文字列としての DNA を解析する最も基本的な方法は，2 つの配列を比較することである．配列が似ていれば機能も似ているという経験的な知識に基づくものである．似たものを探すための基本は，並べて比較してみることである．このように並べて似ているかどうかを調べる操作のことを，アライメント (alignment) という．2 つの配列をただ並べるのではなく，似ている部分同士が一致するように並べる．

例えば，次の 2 つの配列において，上行にある DNA 配列が実験から得られたとき，これを既に機能などが解析されている下行の DNA 配列と比較することを考える：

```
GAGGTTATCAAAAGCTACTAGTCCA   (実験から得られた配列)
GAGGATAACAAGGCTACTATCACA    (機能などが既知の配列)
```

この 2 つの配列をなるべく多くの文字が一致するように並べると次のようになる：

```
GAGGTTATCAA-AA-GCTACTAGTC-CA
GAGG--AT-AACAAGGCTACTA-TCACA
```

このように，2 つの配列を並べる時に両者が一致しない場所がある場合，その場所を空として飛び越して次の配列をつなげる必要がある．この空の場所のことを「ギャップ」と呼ぶ（上の図では，“-” という記号を入れて表している）．ギャップは，生物学的には，進化の過程において突然変異などによって，その部分に新たに挿入されたり，または欠失したりしたものと解釈される．また，他の塩基に置き換わる置換も起こり得る．この挿入・欠失・置換の操作によって，ある配列から別の配列へ変換するのに必要とする操作の最小ステップ数のことを 2 つの配列間の編集距離と呼ぶ．

ペアワイズアライメントとは，2 つの DNA 配列に対して適切な位置にギャップ記号を挿入することによって，配列中の同じ位置に同じ塩基（あるいは性質がよく似た塩基）が並ぶようにする操作のことである．さらに，マルチプルアラインメントとは，3 本以上の複数の配列に対して，

[2]例えば，大部分がソート済みである配列にこの分割アルゴリズムを適用すると大きな偏りが生じることになる．そして，実際の問題ではこのような場面はよくある．

同じ塩基（あるいは性質がよく似た塩基）ができるだけ同じカラムに並ぶように，適切な位置にギャップ記号を挿入して各配列を並べる操作であり，ペアワイズアライメントを 3 本以上の配列に拡張したものである．

長さ n の配列に対して k 個のギャップを入れる組合せは ${}_nC_k$ 通り存在するので，長さ n の 2 本の配列のアラインメントの作り方は，

$$\sum_{k=1}^{n} {}_nC_k \times {}_nC_k = {}_{2n}C_n \cong \frac{2^{2n}}{\sqrt{\pi n}} \tag{1}$$

程度存在し，すべてのアラインメントを列挙して最適なアラインメントを選択することは n が大きくなると現実的には不可能である [3]．しかし，配列のアラインメントを計算する問題は，求めたいのは最適解のみであること，問題を部分問題に分割して解くことができることから，**動的計画法** (dynamic programming) と呼ばれるアルゴリズムを用いて非常に効率良く求めることができる．

例えば，`AGCG` と `GTCAG` のアラインメントを求める問題は，次のような 3 つの部分問題に分割することができる．

$$\left(\begin{array}{c}\texttt{AGC}\\ \texttt{GTCA}\end{array}\right)\text{の解} + \left[\begin{array}{c}\texttt{G}\\ \texttt{G}\end{array}\right],\quad \left(\begin{array}{c}\texttt{AGC}\\ \texttt{GTCAG}\end{array}\right)\text{の解} + \left[\begin{array}{c}\texttt{G}\\ -\end{array}\right],\quad \left(\begin{array}{c}\texttt{AGCG}\\ \texttt{GTCA}\end{array}\right)\text{の解} + \left[\begin{array}{c}-\\ \texttt{G}\end{array}\right]$$

それぞれの $\left(\begin{array}{c}\\ \\ \end{array}\right)$ について最適解がわかっていれば，それらに [ ] の部分を付け加えて伸長した時の最適解を計算することは容易であり，これらの 3 つの中で最適なものを `AGCG` と `GTCAG` の最適アラインメントとすればよい．それぞれの部分問題は同様にしてさらに細かい部分問題に分割していく．同じ部分問題が出てきた時に再計算することを避けるために，すべての部分問題の最適解を表に記録していく．

この考えは以下のようなアルゴリズムで実現することができる．長さ m の配列 $\texttt{x} = x_1x_2\cdots x_m$ と長さ n の配列 $\texttt{y} = y_1y_2\cdots y_n$ を入力とする．`dp` と `trace` を大きさ $(m+1)\times(n+1)$ の 2 次元配列とする．ここで，`dp` は 2 つの配列間の類似度を計算するための 2 次元配列であり，`trace` はアライメントを生成するための 2 次元配列である．2 次元配列 `dp` は，次のように再帰的に定義される：

$$\texttt{dp[}i\texttt{][}j\texttt{]} = \max\begin{cases}\texttt{dp[}i-1\texttt{][}j-1\texttt{]} + s(x_i, y_j), & -\text{ case (1)},\\ \texttt{dp[}i-1\texttt{][}j\texttt{]} + d, & -\text{ case (2)},\\ \texttt{dp[}i\texttt{][}j-1\texttt{]} + d, & -\text{ case (3)},\end{cases} \tag{2}$$

$$\text{初期化：}\ \texttt{dp[0][0]} = 0,\ \texttt{dp[}i\texttt{][0]} = i\times d,\ \texttt{dp[0][}j\texttt{]} = j\times d. \tag{3}$$

ここで，$s(x_i, y_j)$ は 2 つの文字 x_i と y_j の間の置換度を表す．核酸の場合には，

$$s(x,y) = \begin{cases}+1 & \text{if } x = y,\\ -1 & \text{if } x \neq y.\end{cases} \tag{4}$$

を用いることが多い．d はギャップを挿入するときのギャップスコアを表し，核酸の場合には $d = 0$ とすることが多い．

[3] $n = 10$ の場合には ${}_{20}C_{10} \cong 1.9\times10^6$，$n = 100$ の場合には ${}_{200}C_{100} \cong 9.1\times10^{58}$ である．ちなみに，地球の年齢は約 46 億年 $\cong 1.4\times10^{17}$ 秒，宇宙の年齢は約 137 億年 $\cong 4.3\times10^{17}$ 秒であるから，$n = 100$ の場合には，宇宙開闢以来，1 秒間につき 100 億 $(= 10^{10})$ 個のアラインメントを計算してきたとしても，いまだに計算が終わっていないことになる．

	j	0	1	2	3	4	5	6
i			G	T	C	A	G	A
0		0	0 ←	0 ←	0 ←	0 ←	0 ←	0 ←
1	A	**0** ↑	0 ↑	0 ↑	0 ↑	1 ↖	1 ←	1 ↖
2	G	0 ↑	**1** ↖	**1** ←	1 ←	1 ↑	2 ↖	2 ←
3	C	0 ↑	1 ↑	1 ↑	**2** ↖	**2** ←	2 ↑	2 ↑
4	G	0 ↑	1 ↖	1 ↑	2 ↑	2 ↑	**3** ↖	3 ←
5	T	0 ↑	1 ↑	2 ↖	2 ↑	2 ↑	**3** ↑	3 ↑
6	A	0 ↑	1 ↑	2 ↑	2 ↑	3 ↖	3 ↑	**4** ↖
7	G	0 ↑	1 ↖	2 ↑	2 ↑	3 ↑	4 ↖	**4** ↑

図 5: 配列 AGCGTAG と GTCAGA を アラインメントした時の dp と trace の値

また，2 次元配列 trace は，式 (2) において選択された case に応じて，

$$\texttt{trace[}i\texttt{][}j\texttt{]} = \begin{cases} \text{“}\nwarrow\text{”} & \text{if case (1)}, \\ \text{“}\uparrow\text{”} & \text{if case (2)}, \\ \text{“}\leftarrow\text{”} & \text{if case (3)}. \end{cases} \tag{5}$$

初期化： $\texttt{trace[}i\texttt{][0]} = \text{“}\uparrow\text{”}$, $\texttt{trace[0][}j\texttt{]} = \text{“}\leftarrow\text{”}$.

のように定義される．

2 つの短い配列 AGCGTAG と GTCAGA をアラインメントした時の dp と trace の値は図 5 のようになり，アラインメント結果は次のようになる．

```
AG-C-GTAG
-GTCAG-A-
```

以上に述べたアルゴリズムを疑似コードで表すと，リスト 4 のようになる．また，trace からアラインメントを得るためには，リスト 5 の疑似コードのような再帰呼び出しを用いればよい．

配列の長さが 2 本とも n 程度だとすると，動的計画法による配列のアラインメントの計算量は $O(n^2)$ であることがリスト 4 からわかる．すべてのアラインメントを数え上げた場合の計算量は式 (1) から $O(2^n)$ であるので，動的計画法の方がはるかに高速であると言える．しかし，動的計画法の場合には，部分問題の結果を記録する表のために $O(n^2)$ の記憶領域が必要となり，記憶領域を犠牲にして計算時間を少なくしていると考えることができる．このように，一般に計算時間と記憶領域はトレードオフの関係にある．

問題 8　リスト 4 とリスト 5 を参考にして配列のアラインメントを計算するプログラムを作りなさい．

参考文献

[1] C で書くアルゴリズム，サイエンス社（ISBN: 4–7819–0790–3）

[2] 岩波講座ソフトウェア科学 アルゴリズムとデータ構造，岩波書店（ISBN: 4–0001–0343–1）

[3] バイオプログラミング第 2 ホームページ（http://www.dna.bio.keio.ac.jp/lecture/progen2/）

リスト 4: 配列間の類似度を計算する align() の疑似コード

```
1  void align(char x[], char y[], int len_x, int len_y)
2  {
3    式 (3) に従って dp と tbl を初期化する.
4
5    for (j=1; j<len_y; j++) {
6      for (i=1; i<len_x; i++) {
7        dp[i-1][j-1]+s(x[i],y[j]), dp[i-1][j]+d, dp[i][j-1]+d の中で
8        最大となるものを選び, dp[i][j] に入れる(式 (2)).
9        trace[i][j] には選んだ方向に対応する矢印 ("↖", "↑", "←") を入れる.
10     }
11   }
12 }
```

リスト 5: アライメントを出力する traceback() の疑似コード

```
1  void traceback(char x[], char y[], int i, int j)
2  {
3    if (i==0 && j==0) {
4      res_x と res_y を空文字列にする.
5    } else if (trace[i][j] == "↖") {
6      traceback(i-1, j-1);
7      res_x に x[i] を加える.
8      res_y に y[j] を加える.
9    } else if (trace[i][j] == "↑") {
10     traceback(i-1, j);
11     res_x に x[i] を加える.
12     res_y に "-" を加える.
13   } else if (trace[i][j] == "←") {
14     res_x に "-" を加える.
15     res_y に y[j] を加える.
16   }
17 }
```