```
0001: !*************************************************************************
0002: !****      整列（sort）と 2分探索（binary search）のルーチン集      ****
0003: !*************************************************************************
0004: !
0005: ! 整列アルゴリズムの分類
0006: !
0007: ! ○ O(n^2)系：実用にならないが本モジュールで実装
0008: !     ・基本選択法：   学習用途のみ
0009: !     ・基本挿入法：   n=数10程度ならば若干速い．ほとんど整列済みデータならば相当速い
0010: !
0011: ! ○ O(n*log(n))系：処理が複雑なので，本モジュールでは未実装
0012: !     ・クイックソート：  一般に最速ゆえに王道．しかし，最悪O(n^2) と要スタック記憶が問題
0013: !                         各種テクが必要 -> 再帰が深いとヒープソートに切替，
0014: !                                        -> 非再帰化
0015: !                                        -> 終盤で挿入法を併用
0016: !     ・ヒープソート：平均速さはクイックソートに劣る．しかし，O(n*log(n))保障，スタック不要
0017: !                       （選択法の応用）
0018: !                       （ヒープ：木を配列で表現したもの，優先度付き待ち行列で多用される）
0019: !
0020: ! ○ その他：本モジュールで実装
0021: !     ・シェルソート：O(n^1.25)  nが数万までなら十分実用的（挿入法の応用）
0022: !
0023: !                       参考文献：石畑清「アルゴリズムとデータ構造」，p149～197，岩波書店
0024: !
0025: !*****************
0026: ! 2分探索
0027: !
0028: !     整列済みかつ無重複のデータ列から指定した値の有無と，その記憶位置もしくは挿入すべき位置
0029: !     を探す．2分探索は，半分ずつ存在範囲を絞り込んでいくので，計算量は O(log(n))．
0030: !     一度のみの探索であれば，データ列の整列が不要な線形探索のほうが良い．線形探索の計算量は
0031: !     O(n)と多いが，整列 ― どんなに速くても O(n*log(n)) ― が不要なぶん，有利である．
0032: !     しかし，何度も探索する場合には，一度手間をかけて整列させてから 2分探索するほうが良い．
0033: !     なお，データの挿入には線形探索が O(1)，2分探索が O(n) ，削除には線形探索が O(n)，
0034: !     2分散策も O(n) である．よって，頻繁なデータ操作がある場合には，別案を考える必要がある．
0035: !
0036: !                       参考文献：石畑清「アルゴリズムとデータ構造」，p67～72，岩波書店
0037: !
```

```
0038:
0039: !*****************
0040: ! 補足）仮引数(かり-ひきすう）としての配列の記述法と，計算効率について
0041: !
0042: ! ○ 形状明示配列(explicit shape array)  ― 副プログラム内の仮引数仕様で a(n) ―
0043: !     ・F77互換のため，仮配列に割付けられたメモリ域は連続である.
0044: !     ・長所：最適化に有利
0045: !     ・短所：実引数・仮引数の結合において，添字の範囲チェックが困難
0046: !     ・短所：実引数が部分配列で，対応するメモリ域が不連続である場合，F77規約違反.
0047: !             F90規約には違反していないが，原理的に一時配列の生成は不可避
0048: !              (intel Fortran では -check:arg_temp_created で一時配列生成が実行時に判明)
0049: !
0050: ! ○ 形状引継配列(assumed shape array)  ― 副プログラム内の仮引数仕様で a(:) ―
0051: !     ・F90で導入された．仮配列に割付けられたメモリ域は連続でなくとも良い.
0052: !       表には一切出てこないが，裏で配列記述子（上限，下限，ストライドなど）が渡されている.
0053: !     ・長所：実引数・仮引数の結合において，添字の範囲チェックが容易
0054: !     ・長所：引数の数が減るため，引数記述時のミスが減る．可読性が上がる
0055: !     ・長所：原理的に一時配列の生成は不要（ベクトル添字の場合は未確認のまま）
0056: !     ・短所：最適化の程度はコンパイラの出来に大きく依存
0057: !              (intel Fortran Ver.12 での経験：
0058: !               -fast オプションで IPO ― Inter-Procedual Optimization, プロシジャ間での
0059: !               大域的な最適化 ― あたりを掛けると，形状明示配列とほぼ同じ速度がでる.
0060: !               しかし，普通の最適化（プロシジャ毎の最適化）では，幾分速度が落ちる.
0061: !               理由：ストライドの存在ゆえに，アンロールなどの積極的な最適化が困難．)
0062: !
0063: ! 例)
0064: !       SUBROUTINE sub_F77 ( a, n )            ! INTERFACE記述 も MODULE内配置 も不要
0065: !          INTEGER, INTENT(inout) :: a(n)      ! 形状明示配列 (explicit shape array)
0066: !          INTEGER, INTNET(in   ) :: n
0067: !                :   (本例では基本挿入法)
0068: !          END SUBROUTINE sub_F90
0069: !       END SUBROUTINE sub_F77
0070: !
0071: !       MODULE procedures
0072: !       CONTAINS
0073: !          SUBROUTINE sub_F90 ( a )            ! INTERFACE記述 もしくは MODULE内配置が必須
0074: !             INTEGER, INTENT(inout) :: a(:)   ! 形状引継配列 (assumed shape array)
0075: !                :   (本例では基本挿入法)
0076: !          END SUBROUTINE sub_F90
0077: !       END MODULE procedures
0078: !
0079: !       PROGRAM test
0080: !          USE procedures
0081: !                :
0082: !          CALL sub_F77( a, n )       ! F77 ではこの呼び方のみ
0083: !                                     ! 一般に最速
0084: !
0085: !          CALL sub_F77( a(n:1:-2), (n+1)/2 ) ! F77では違反だが，F90では必ず一時配列を作る
0086: !                                     ! 一時配列は基本的にstack域に．大きいとstack overflow
0087: !
0088: !          CALL sub_F90( a )          ! F90 推奨の呼び方
0089: !                                     ! しっかりしたF90コンパイラならば sub_F77 と同じ速さ
0090: !                                     !  (intel Fortran Ver.12 ならば -fast で同じ速さに.
0091: !                                     !   逆に，何もオプションなければ数割遅かった.
0092: !                                     !   結局，IPO などで 1以外のストライドの可能性の有無を
0093: !                                     !   コンパイラが適切に処理できるか否かにある)
0094: !
0095: !          CALL sub_F90( a(n:1:-2) ) ! しっかりしたF90コンパイラならば 一時配列を作らない
0096: !                                     ! (ストライド必須ゆえ，intel Fortran Ver.12 で
0097: !                                     !   -fast を指定しても数割遅いまま．これは納得できる)
0098: !       END PROGRAM test
0099: !
0100: !       参考文献：配列引数の効率的な渡し方
0101: !                http://www.xlsoft.com/jp/products/intel/cvf/docs/vf-html/pg/pg06_04_02.htm
0102: !
0103: !
```

```
0104:
0105: MODULE sort_and_search_module
0106:
0107:    IMPLICIT none
0108:    PRIVATE
0109:
0110:    PUBLIC sort
0111:    INTERFACE sort
0112:     ! MODULE PROCEDURE selection_sort_int          ! 学習用
0113:     ! MODULE PROCEDURE insertion_sort_int          ! 学習用
0114:       MODULE PROCEDURE shells_sort_int, &
0115:                      & shells_sort_int_with_priority
0116:    END INTERFACE
0117:
0118:    PUBLIC binary_search
0119:    INTERFACE binary_search
0120:       MODULE PROCEDURE binary_search_method_1_int  ! いずれか1つのみを選択
0121:     ! MODULE PROCEDURE binary_search_method_2_int  !
0122:    END INTERFACE
0123:
0124: CONTAINS
0125:
0126: !
```

```
0127:
0128: !*************************************************************************************
0129:    SUBROUTINE selection_sort_int ( a )
0130:
0131:       ! 学習用途：単純選択法による a(:) の昇順整列
0132:
0133:       !     比較回数：n*(n-1)/2
0134:       !     交換回数：最大 n-1
0135:
0136:       INTEGER, INTENT(inout) :: a(:)
0137:       INTEGER :: i, j, n
0138:       INTEGER :: m                    ! a(:) 中で最小値候補がある場所
0139:       INTEGER :: tmp
0140:
0141:       n = SIZE(a)
0142:
0143:       DO i = 1, n-1
0144:          ! a(1:i-1) は既に最終形の昇順整列にあり, a(i:n) のみが興味の対象.
0145:          ! a(i:n) から最小値を探して, その位置を m とすれば, a(m) と a(i)
0146:          ! を入れ替えて, a(1:i) までが最終形の昇順整列を得る.
0147:
0148:          ! 最小値の位置を探す
0149:          m = i                        ! a(i:i)中の最小値の位置
0150:          DO j = i+1, n
0151:             IF ( a(m) > a(j) ) m = j  ! a(i+1:j)中の最小値の位置
0152:          END DO
0153:          ! a(i) の値と a(m) の値を入れ替え
0154:          IF ( m /= i ) THEN
0155:             tmp  = a(i)
0156:             a(i) = a(m)
0157:             a(m) = tmp
0158:          END IF
0159:
0160:       END DO
0161:
0162:       ! 例)  a(:)
0163:       !    |_4_|_2_|_1_|_3_|_9_|_2_|
0164:       !
0165:       ! i=1  v -->--->--->--->--->   m = 1  -> 2  -> 3  -> 3  -> 3  -> 3
0166:       !    |_4_|_2_|_1_|_3_|_9_|_2_|
0167:       !                ^             swap( a(i), a(m) )
0168:       !    |_1_|_2_|_4_|_3_|_9_|_2_|
0169:       !
0170:       ! i=2 .済. v -->--->--->--->         m = 2  -> 2  -> 2  -> 2  -> 2
0171:       !    |_1_|_2_|_4_|_3_|_9_|_2_|
0172:       !            ^                 swap( a(i), a(m) ) 不要
0173:       !    |_1_|_2_|_4_|_3_|_9_|_2_|
0174:       !
0175:       ! i=3 ...済... v -->--->--->               m = 3  -> 4  -> 4  -> 6
0176:       !    |_1_|_2_|_4_|_3_|_9_|_2_|
0177:       !                        ^     swap( a(i), a(m) )
0178:       !    |_1_|_2_|_2_|_3_|_9_|_4_|
0179:       !
0180:       ! i=4 .....済..... v -->--->                     m = 3  -> 3  -> 3
0181:       !    |_1_|_2_|_2_|_3_|_9_|_4_|
0182:       !                    ^         swap( a(i), a(m) ) 不要
0183:       !    |_1_|_2_|_2_|_3_|_9_|_4_|
0184:       !
0185:       ! i=5 .......済....... v -->                           m = 5  -> 6
0186:       !    |_1_|_2_|_2_|_3_|_9_|_4_|
0187:       !                        ^     swap( a(i), a(m) )
0188:       !    |_1_|_2_|_2_|_3_|_4_|_9_|
0189:
0190:    END SUBROUTINE selection_sort_int
0191:
0192: !
```

```
0193:
0194:    SUBROUTINE insertion_sort_int ( a )
0195:
0196:       ! 学習用途：単純挿入法による a(:) の昇順整列
0197:
0198:       !     比較回数：最大 n*(n-1)/2
0199:       !     交換回数：最大 n*(n-1)/2
0200:       !     特記事項：ほとんど整列済みのデータに対しては高速
0201:
0202:       INTEGER, INTENT(inout) :: a(:)
0203:       INTEGER :: i, j, n
0204:       INTEGER :: pvt                    ! 注目する a(i) の値（値の退避場所を兼用）
0205:
0206:       n = SIZE(a)
0207:
0208:       DO i = 2, n
0209:          ! a(1:i-1) は最終形でないが とりあえず昇順整列状態にある．そこで，a(i)の値に注目
0210:          pvt = a(i)
0211:          IF ( a(i-1) > pvt ) THEN       ! 該当しない場合は最初から整列済み（幸運）
0212:             ! pvtの値をa(1:i)の昇順となる位置に挿入（挿入ケ所を空けるため右から順に右に繰る）
0213:             a(i) = a(i-1)               ! ループの前処理： 1つ右に繰る
0214:             DO j = i-1, 2, -1           ! EXITせずにDOループを終えると j=1 に（Fortran仕様）
0215:                IF ( a(j-1) <= pvt ) EXIT ! 昇順となる位置jを発見
0216:                a(j) = a(j-1)            ! a(1:i-2) の値を 右から順に1つ右に繰る
0217:             END DO
0218:             a(j) = pvt
0219:          END IF
0220:       END DO
0221:
0222:       ! 例)  a(:)                        pvt（レジスタ）
0223:       !    |_4_|_2_|_1_|_3_|_9_|_2_|  [ x ]
0224:       !
0225:       ! i=2      v                          a(2)の値を とりあえず pvtにコピー
0226:       !    |_4_|_2_|_1_|_3_|_9_|_2_|  [(2)]
0227:       !       -->                         a(1)> pvt なので      a(1)の値を a(2)に上書き
0228:       !    |_x_|_4_|_1_|_3_|_9_|_2_|  [ 2 ]
0229:       !                                  左端に達したので      pvt の値を a(1)に挿入
0230:       !    |_2_|_4_|_1_|_3_|_9_|_2_|  [ x ]
0231:       !
0232:       ! i=3         v                       a(3)の値を とりあえず pvtにコピー
0233:       !    |_2_|_4_|_1_|_3_|_9_|_2_|  [(1)]
0234:       !          -->                       a(2)> pvt なので      a(2)の値を a(3)に上書き
0235:       !    |_2_|_x_|_4_|_3_|_9_|_2_|  [ 1 ]
0236:       !       -->                         a(1)<=pvt でないので  a(1)の値を a(2)に上書き
0237:       !    |_x_|_2_|_4_|_3_|_9_|_2_|  [ 1 ]
0238:       !                                  左端に達したので       pvtの値を a(1)に挿入
0239:       !    |_1_|_2_|_4_|_3_|_9_|_2_|  [ x ]
0240:       !
0241:       ! i=4             v                  a(4)の値を とりあえず pvtにコピー
0242:       !    |_1_|_2_|_4_|_3_|_9_|_2_|  [(3)]
0243:       !             -->                    a(3)> pvt なので      a(3)の値を a(4)に上書き
0244:       !    |_1_|_2_|_x_|_4_|_9_|_2_|  [ 3 ]
0245:       !                                  a(2)<=pvt なので      pvt の値を a(3)に挿入
0246:       !    |_1_|_2_|_3_|_4_|_9_|_2_|  [ x ]
0247:       !
0248:       ! i=5                 v             a(5)の値を とりあえず pvtにコピー
0249:       !    |_1_|_2_|_3_|_4_|_9_|_2_|  [(9)]
0250:       !                                  a(4)> pvt でないので  何も処理せず 次へ
0251:       !    |_1_|_2_|_3_|_4_|_9_|_2_|  [ x ]
0252:       !
0253:       ! i=6                     v         a(6)の値を とりあえず pvtにコピー
0254:       !    |_1_|_2_|_3_|_4_|_9_|_2_|  [(2)]
0255:       !                       -->          a(5)> pvt なので      a(5)の値を a(6)に上書き
0256:       !    |_1_|_2_|_3_|_4_|_x_|_9_|  [ 2 ]
0257:       !                   -->              a(4)<=pvt でないので  a(4)の値を a(5)に上書き
0258:       !    |_1_|_2_|_3_|_x_|_4_|_9_|  [ 2 ]
0259:       !               -->                  a(3)<=pvt でないので  a(3)の値を a(4)に上書き
0260:       !    |_1_|_2_|_x_|_3_|_4_|_9_|  [ 2 ]
0261:       !                                  a(2)<=pvt なので      pvt の値を a(3)に挿入
0262:       !    |_1_|_2_|_2_|_3_|_4_|_9_|  [ x ]
0263:
0264:    END SUBROUTINE insertion_sort_int
0265:
0266: !
```

```
0267:
0268:     SUBROUTINE shells_sort_int ( a )
0269:
0270:       ! シェルソート(Shell's sort)による a(:) の 昇順整列
0271:       !
0272:       !    単純挿入法がほとんど整列済みのデータ列に対して高速なことを利用した整列法.
0273:       !    1本のデータ列をh本のデータ列に細分すれば, 各列の単純挿入には O((n/h)^2).
0274:       !    h本あるので, 合計でも O(n^2/h) に過ぎない. そこで, この h を ..., 40, 13, 4, 1
0275:       !    などと漸減させていくことで, 多段的に大雑把な整列をする. h>1 が前処理工程であり,
0276:       !    最終工程 h=1 は単純挿入である.
0277:       !
0278:       ! 刻み h は素数列が良いが, 漸化式を h ← 3*h +1 とした場合, 経験的に 平均して O(n^1.25)
0279:       ! 例えば, n = 10000 ならば n*log(n)=13*10000 > n^1.25 = 10*10000. O(n*log(n))系より速い
0280:
0281:       INTEGER,            INTENT(inout) :: a(:)
0282:
0283:       INTEGER :: i, j, n
0284:       INTEGER :: pvt                    ! 注目する a(i) の値 (値の退避場所を兼用)
0285:       INTEGER :: h                      ! 刻み ( h>1 は前処理工程,  h=1 は単純挿入工程)
0286:
0287:       n = SIZE(a)
0288:
0289:       ! n を超えない最大の刻み h を求める
0290:       h = 1
0291:       DO WHILE ( h < n )
0292:          h = 3*h +1        !  h = 1, 4, 13, 40, ...
0293:       END DO
0294:
0295:       DO WHILE ( h > 1 )
0296:          h = h/3 +1     !  h = ..., 40, 13, 4, 1
0297:          ! h>1 は 前処理工程, 最後の h=1 は 最終の単純挿入工程
0298:
0299:          DO i = 1+h, n
0300:             ! h本のデータ列 a(1:i-h:h), a(2:i-h+1:h), ... , a(h:i-1:h) は,
0301:             ! それぞれ昇順整列状態にある (各列同士は無関係) . 対応列の a(i) の値に注目
0302:             pvt = a(i)
0303:             IF ( a(i-h) > pvt ) THEN         ! 該当しない場合は最初から整列済み (幸運)
0304:                ! pvtの値を 対応列で昇順となる位置に挿入
0305:                a(i) = a(i-h)                 ! ループの前処理 : +h右に繰る(対応列では+1右)
0306:                DO j = i-h, 1+h, -h           ! EXITせずにDOループを終えると 1<=j<=h に
0307:                   IF ( a(j-h) <= pvt ) EXIT ! 昇順となる位置jを発見
0308:                   a(j) = a(j-h)             ! +h右に繰る(対応列では+1右)
0309:                END DO
0310:                a(j) = pvt
0311:             END IF
0312:          END DO
0313:
0314:       END DO
0315:
0316:     END SUBROUTINE shells_sort_int
0317:
0318: !
```

```
0319:
0320:    SUBROUTINE shells_sort_int_with_priority ( a, pri )
0321:
0322:       ! 優先度 pri(:) をキーとする a(:) の昇順整列（参考：優先度付き待ち行列）
0323:
0324:       !    実装はシェルソート(Shell's sort)   （いずれはヒープを使いたいかも…）
0325:
0326:       INTEGER,            INTENT(inout) :: a   (:)
0327:       INTEGER,            INTENT(inout) :: pri(:)  ! 優先度
0328:
0329:       INTEGER :: i, j, n
0330:       INTEGER :: pvt, tmp
0331:       INTEGER :: h
0332:
0333:       n = SIZE(a)
0334:       IF ( n /= SIZE(pri) ) STOP "sort: データと優先度の個数が不一致"
0335:
0336:       h = 1
0337:       DO WHILE ( h < n )
0338:          h = 3*h +1
0339:       END DO
0340:
0341:       DO WHILE ( h > 1 )
0342:          h = h/3 +1
0343:
0344:          DO i = 1+h, n
0345:             pvt = pri(i)
0346:             IF ( pri(i-h) > pvt ) THEN
0347:                                                ; tmp  = a(i)
0348:                pri(i) = pri(i-h)               ; a(i) = a(i-h)
0349:                DO j = i-h, 1+h, -h
0350:                   IF ( pri(j-h) <= pvt ) EXIT
0351:                   pri(j) = pri(j-h)            ; a(j) = a(j-h)
0352:                END DO
0353:                pri(j) = pvt                    ; a(j) = tmp
0354:             END IF
0355:          END DO
0356:
0357:       END DO
0358:
0359:    END SUBROUTINE shells_sort_int_with_priority
0360:
0361: !
```

```
0362:
0363: !*****************************************************************************************
0364:    FUNCTION binary_search_method_1_int ( j, a, exist_ ) RESULT ( loc )
0365:
0366:       ! 2分探索法（方法1）により，昇順整列済み無重複データ a(:) 中における jの値を探索
0367:       ! j の値が存在すれば その位置 loc を，存在しなければ 挿入すべき位置 loc
0368:       !  ( loc 以降を1つ後退： a(loc+1:n+1) = a(loc:n); a(loc) = j ) を返す.
0369:
0370:       ! 理解し易いであろうアルゴリズム版
0371:       !    ・短所：ループ中に 条件分岐が 2 つ
0372:       !    ・長所：存在すれば log(n) 回以下のループで済むことが多い
0373:
0374:       INTEGER,             INTENT(in   ) :: j       ! 探索値
0375:       INTEGER,             INTENT(in   ) :: a(:)    ! 昇順整列済み無重複データ
0376:       LOGICAL, OPTIONAL,   INTENT(out  ) :: exist_  ! 探索値の存在の有無
0377:       INTEGER                            :: loc     ! a(:) 中の位置
0378:
0379:       INTEGER :: left, middle, right
0380:       LOGICAL :: it_exists
0381:
0382:       left  = LBOUND(a,1)
0383:       right = UBOUND(a,1)
0384:       DO WHILE ( left <= right )
0385:          middle = ( left + right ) / 2
0386:          IF ( j <= a(middle) ) right = middle -1  ! j は真中もしくはそれより左ゆえ，右を詰める
0387:          IF ( a(middle) <= j ) left  = middle +1  !     〃          〃   右  〃  左   〃
0388:          ! j == a(middle) の場合，両側から一気に狭まり，左右逆転してループ終了
0389:          ! （したがって，2肢択一でなく 3肢択一 となっていることに注意）
0390:       END DO
0391:       ! ここに辿り着いたときには，a(right) < j < a(left) となっており，
0392:       !     j の値が あった ときには，left - right = 2  ( j = a(middle) の両隣 )
0393:       !       〃    なかった   〃     left - right = 1  ( 存在しない値j の両隣 )
0394:
0395:       it_exists = ( left - right == 2 )
0396:       IF ( it_exists ) THEN
0397:          loc = middle
0398:       ELSE
0399:          loc = left
0400:       END IF
0401:
0402:       IF ( PRESENT(exist_) )  exist_ = it_exists
0403:
0404:       ! 例)     1   2   3   4   v   6    =====  j=8 を探索  =====
0405:       !  a(:)=|_2_|_3_|_4_|_6_|_8_|_9_|  1) left = 1, right = 6 からはじめる
0406:       !     left|       |           |right  middle = (left + right )/2 = 3
0407:       !                 middle              a(middle) = 4 < j ゆえに，left  = middle +1 = 4
0408:       !                                 2)
0409:       !                   left|   |   |right  middle = (left + right )/2 = 5
0410:       !                         middle        a(middle) = 8 = j ゆえに，right = 4, left = 6
0411:       !                                 3)
0412:       !                  right|       |left   もはや left <= right でないので終了
0413:       !                                       なお，j の値の位置は middle が指している
0414:       !                                    ― 見つかる場合は，2等分割の途中が多い ―
0415:       !
0416:       !           1   2   3   4 v 5   6    =====  j=7 を探索  =====
0417:       !  a(:)=|_2_|_3_|_4_|_6_|_8_|_9_|  1) left = 1, right = 6 からはじめる
0418:       !     left|       |           |right  middle = (left + right )/2 = 3
0419:       !                 middle              a(middle) = 4 < j ゆえに，left  = middle +1 = 4
0420:       !                                 2)
0421:       !                   left|   |   |right  middle = (left + right )/2 = 5
0422:       !                         middle        a(middle) = 8 > j ゆえに，right = middle -1 = 4
0423:       !                                 3)
0424:       !                   left|right          middle = (left + right )/2 = 4
0425:       !                     middle            a(middle) = 6 < j ゆえに，left  = middle +1 = 5
0426:       !                                 4)
0427:       !                  right|   |left       もはや left <= right でないので終了
0428:       !                                       なお，j の値を挿入すべき位置は left が指している
0429:       !                                    ― 見つからない場合は，必ず2等分割の最後まで ―
0430:       !
0431:       !       v 1   2   3   4   5   6    =====  j=0 を探索  =====
0432:       !  a(:)=|_2_|_3_|_4_|_6_|_8_|_9_|  1) left = 1, right = 6 からはじめる
0433:       !     left|       |           |right  middle = (left + right )/2 = 3
0434:       !                 middle              a(middle) = 4 > j ゆえに，right = middle -1 = 2
0435:       !                                 2)
0436:       !     left|   |right                  middle = (left + right )/2 = 1
0437:       !       middle                        a(middle) = 2 > j ゆえに，right = middle -1 = 0
```

```
0438:       !                                   3)
0439:       !right|   |left                        もはや left <= right でないので終了
0440:       !                                      なお, j の値を挿入すべき位置は left が指している
0441:       !                                   ― 見つからない場合は, 必ず2等分割の最後まで ―
0442:
0443:     ! ! 以下の書き方もある（3肢択一なので同じこと）
0444:     ! DO WHILE ( left <= right )
0445:     !    middle = ( left + right ) / 2
0446:     !    IF      ( a(middle) == j ) THEN  ! jを発見したので, 終了
0447:     !       loc   = middle
0448:     !       RETURN
0449:     !    ELSE IF ( a(middle)  < j ) THEN  ! j は真中より右ゆえ, 左を詰める
0450:     !       left  = middle +1
0451:     !    ELSE                             !      〃     左  〃  右   〃
0452:     !       right = middle -1
0453:     !    END IF
0454:     ! END DO
0455:     ! ! a(:)中に jの値が見つかれば, ここにたどり着くことはない.
0456:     ! loc = left
0457:
0458:     END FUNCTION binary_search_method_1_int
0459:
0460:
0461:     FUNCTION binary_search_method_2_int ( j, a, exist_ ) RESULT ( loc )
0462:
0463:       ! 2分探索法（方法2）により, 昇順整列済み無重複データ a(:) 中における jの値を探索
0464:       ! j の値が存在すれば その位置 loc を, 存在しなければ 挿入すべき位置 loc
0465:       ! （ loc 以降を1つ後退： a(loc+1:n+1) = a(loc:n); a(loc) = j ）を返す.
0466:
0467:       ! 理解し辛いであろうアルゴリズム版
0468:       !   ・長所：ループ中に 条件分岐が 1 つ
0469:       !   ・短所：必ず log(n) 回のループが必要
0470:
0471:       INTEGER,            INTENT(in   ) :: j       ! 探索値
0472:       INTEGER,            INTENT(in   ) :: a(:)    ! 昇順整列済み無重複データ
0473:       LOGICAL, OPTIONAL,  INTENT(out  ) :: exist_  ! 探索値の存在の有無
0474:       INTEGER                           :: loc     ! a(:) 中の位置
0475:
0476:       INTEGER :: left, middle, right
0477:
0478:       left  = LBOUND(a,1)
0479:       right = UBOUND(a,1)
0480:       DO WHILE ( left <= right )
0481:          middle = ( left + right ) / 2
0482:          IF ( a(middle) < j ) THEN        ! j は真中より       右ゆえ, 左を詰める
0483:             left  = middle +1
0484:          ELSE                             ! j は真中かそれより左    〃 右   〃
0485:             right = middle -1
0486:          END IF
0487:       END DO
0488:       loc = left
0489:
0490:       IF ( PRESENT(exist_) ) THEN
0491:          IF ( loc > UBOUND(a,1) ) THEN
0492:             exist_ = .FALSE.
0493:          ELSE
0494:             exist_ = ( j==a(loc) )
0495:          END IF
0496:       END IF
0497:
0498:     END FUNCTION binary_search_method_2_int
0499:
0500: END MODULE sort_and_search_module
```