Divide_and_Conquer

June 16, 2022

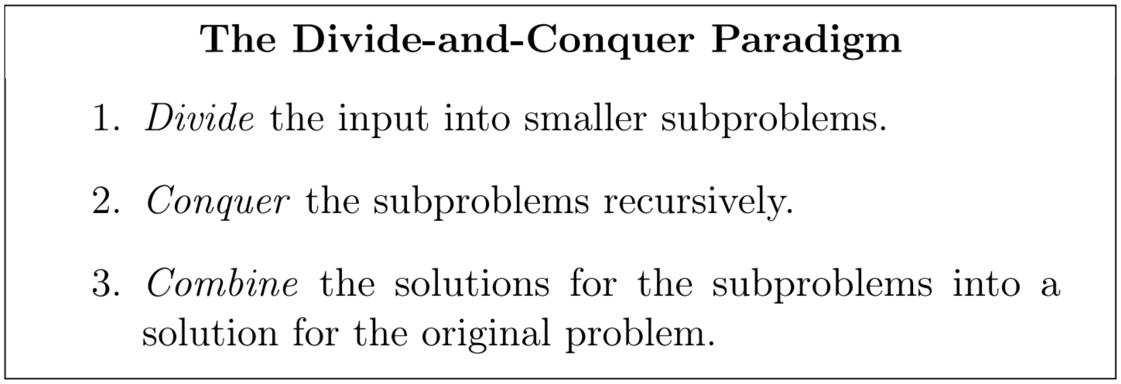
[1]:
```
%%html
<style>
    p,ul {

          font-size: 18px;
          font-family: "Segoe Print","Georgia", "Verdana", "Lucida Console",␣
 ↪"Courier New";
          line-height: 150%;
    }

    .text_cell_render p {
    text-align: justify;
    text-justify: inter-word;
    }
</style>
```

```
<IPython.core.display.HTML object>
```

[2]:
```
from IPython.display import Image
# Image(filename = "gd5.png", width = "100%")
```

[3]:
```
Image(filename = "dc1.png", width = "100%")
```

[3]:

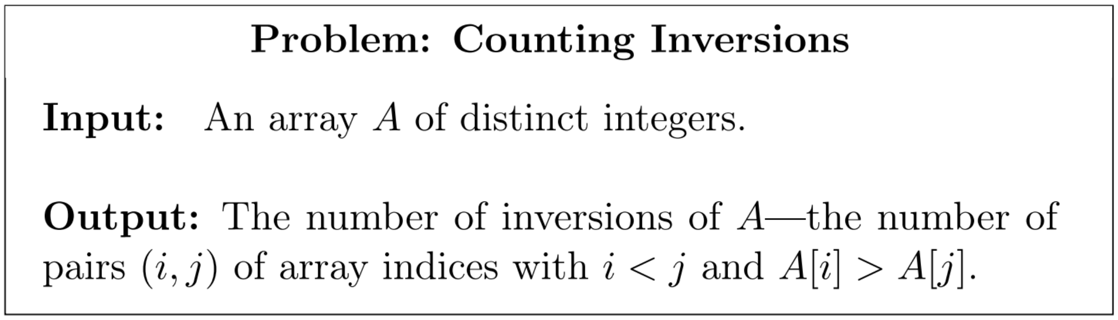


The Divide-and-Conquer Paradigm

1. *Divide* the input into smaller subproblems.
2. *Conquer* the subproblems recursively.
3. *Combine* the solutions for the subproblems into a solution for the original problem.

[4]:
```
Image(filename = "dc2.png", width = "100%")
```

[4]:



Problem: Counting Inversions

Input: An array A of distinct integers.

Output: The number of inversions of A—the number of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

An inversion of an array is a pair of elements that are ``out of order,'' meaning that the element that occurs earlier in the array is bigger than the one that occurs later.

Left inversion: an inversion with i, j both in the first half of the array (i.e., $i, j \leq \frac{n}{2}$)

Right inversion: an inversion with i, j both in the second half of the array (i.e., $i, j > \frac{n}{2}$)

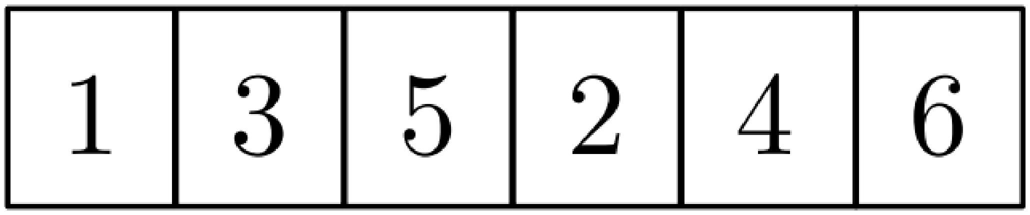
Split inversion: an inversion with i in the left half and j in the right half (i.e., $i \leq \frac{n}{2} < j$)

For example, an array A that is in sorted order has no inversions.

Every array that is not in sorted order has at least one inversion.

[5]:
```
Image(filename = "dc3.png", width = "40%")
```

[5]:



1	3	5	2	4	6

How many inversions does this array have?

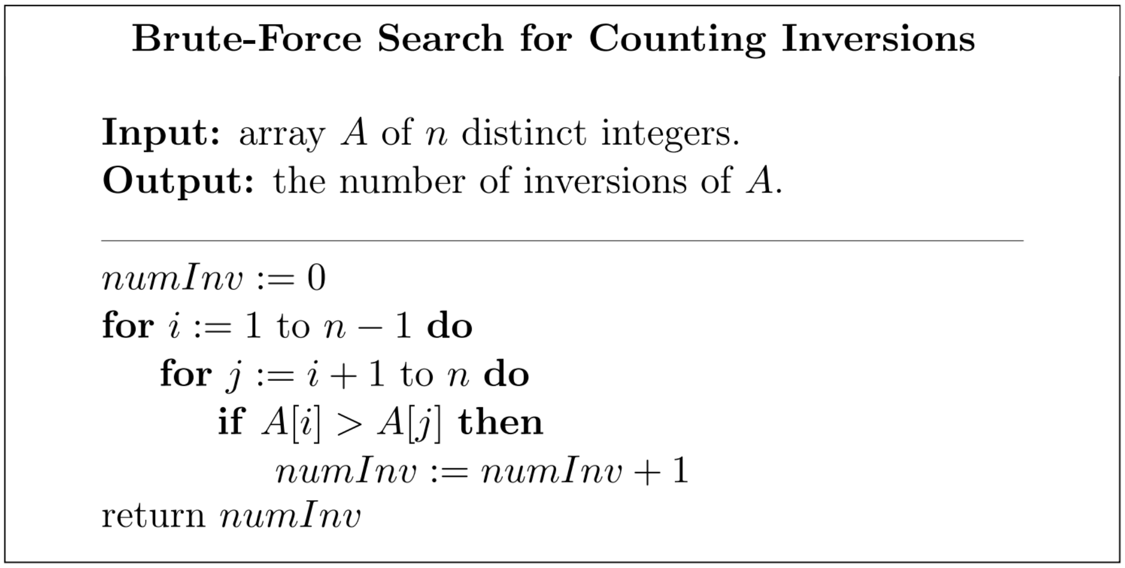
What is the largest-possible number of inversions a 6-element array can have?

Why would you want to count the number of inversions in an array?

- To compute a numerical similarity measure that quantifies how close two ranked lists are to each other.
- One reason you might want a similarity measure between rankings is to do collaborative filtering, a technique for generating recommendations.

```
[6]: Image(filename = "dc4.png", width = "100%")
```

[6]:



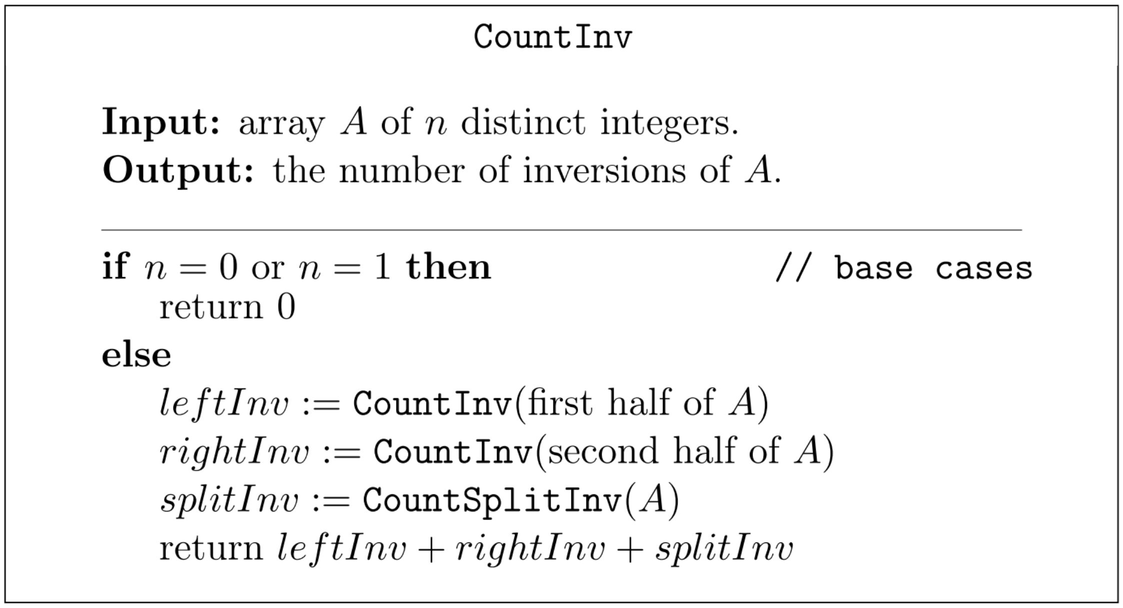
Brute-Force Search for Counting Inversions

Input: array A of n distinct integers.
Output: the number of inversions of A.

```
$numInv := 0$
for $i := 1$ to $n-1$ do
    for $j := i+1$ to $n$ do
        if $A[i] > A[j]$ then
            $numInv := numInv + 1$
return $numInv$
```

```
[7]: Image(filename = "dc5.png", width = "100%")
```

[7]:



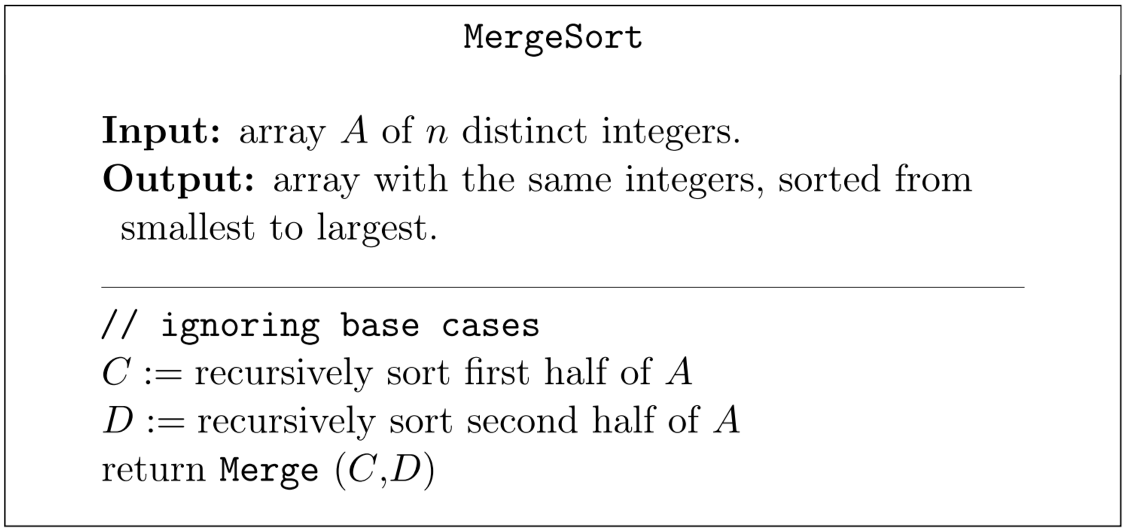
`CountInv`

Input: array A of n distinct integers.
Output: the number of inversions of A.

```
if $n = 0$ or $n = 1$ then                // base cases
    return 0
else
    $leftInv :=$ CountInv(first half of $A$)
    $rightInv :=$ CountInv(second half of $A$)
    $splitInv :=$ CountSplitInv($A$)
    return $leftInv + rightInv + splitInv$
```

```
[8]: Image(filename = "merge2.png", width = "100%")
```

[8]:



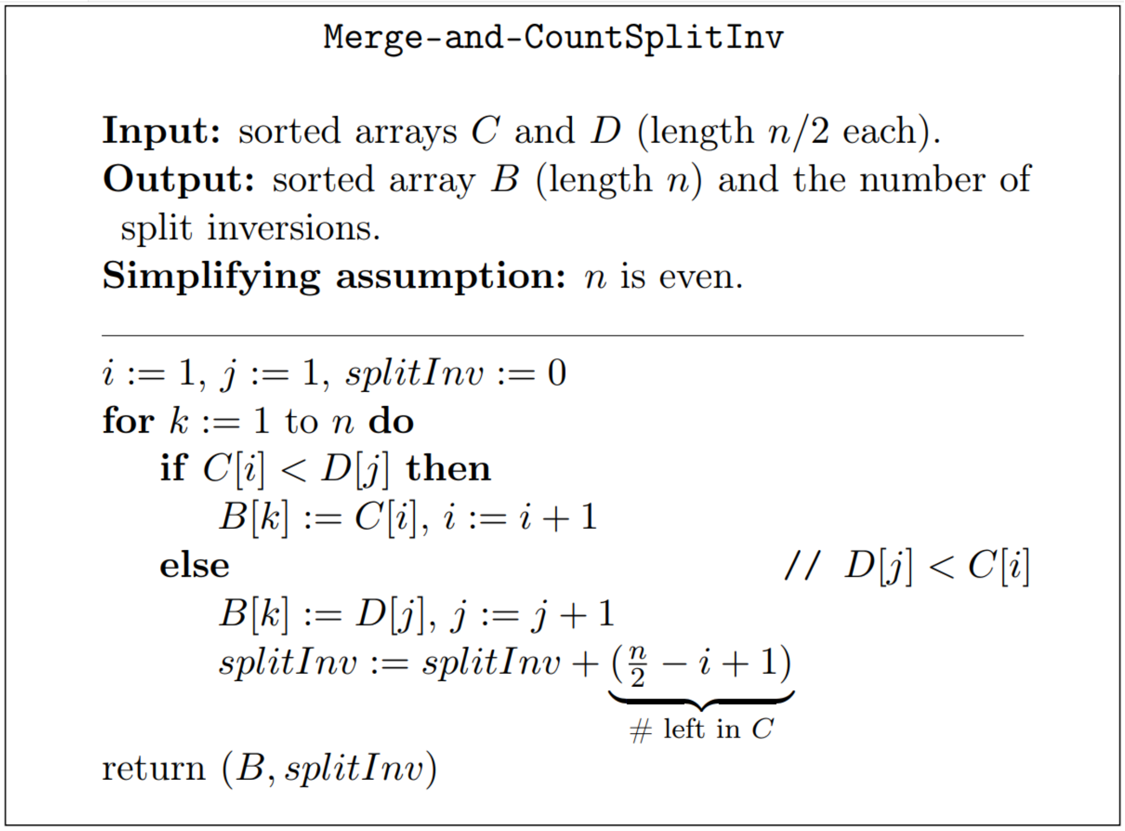
`MergeSort`

Input: array A of n distinct integers.
Output: array with the same integers, sorted from smallest to largest.

```
// ignoring base cases
$C$ := recursively sort first half of $A$
$D$ := recursively sort second half of $A$
return Merge ($C$,$D$)
```

[9]:
```
Image(filename = "dc6.png", width = "100%")
```

[9]:



`Merge-and-CountSplitInv`

Input: sorted arrays C and D (length $n/2$ each).
Output: sorted array B (length n) and the number of split inversions.
Simplifying assumption: n is even.

```
$i := 1$, $j := 1$, $splitInv := 0$
for $k := 1$ to $n$ do
    if $C[i] < D[j]$ then
        $B[k] := C[i]$, $i := i + 1$
    else                                    // $D[j] < C[i]$
        $B[k] := D[j]$, $j := j + 1$
        $splitInv := splitInv + \underbrace{(\frac{n}{2} - i + 1)}_{\#\text{ left in } C}$
return $(B, splitInv)$
```

The maximum-possible number of inversions is at most the number of ways of choosing i, j from the set { 1, 2, . . . , 6 } $ with i < j. There are n(n -

1) ways to choose (i, j) so that i not equals to j (n choices for i, then (n - 1) for j). By symmetry, i < j in exactly half of these.

In an array with no split inversions, everything in the first half is less than everything in the second half.

If A consists of the numbers n/2+1, … , n in order, followed by the numbers 1, 2, … , n/2 in order, there are n^2/4 split inversions.

[ ]:
```
def mergeSortInversions(arr):
    if len(arr) == 1:
        return arr, 0
    else:
        a = arr[:len(arr)//2]
        b = arr[len(arr)//2:]
        a, ai = mergeSortInversions(a)
        b, bi = mergeSortInversions(b)
        c = []
        i = 0
        j = 0
        inversions = 0 + ai + bi
        while i < len(a) and j < len(b):
            if a[i] <= b[j]:
                c.append(a[i])
                i += 1
            else:
                c.append(b[j])
                j += 1
                inversions += (len(a)-i)
        c += a[i:]
        c += b[j:]
    return c, inversions

lst = [1, 3, 5, 2, 4, 6]

sa, ni = mergeSortInversions(lst)

print(ni)
```

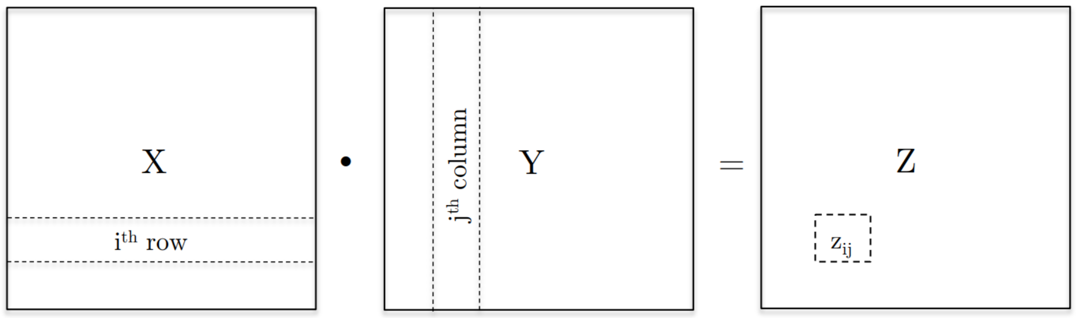
0.1 Matrix Multiplication

Suppose X and Y are $n \times n$ matrices of integers---n^2 entries in each. In the product $Z = X \cdot Y$, the entry z_{ij} in the ith row and jth column of Z is defined as the dot product of the ith row of X and the jth column of Y

[10]:
```
Image(filename = "mm1.png", width = "80%")
```

[10]:

$$z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}.$$



```
[11]: Image(filename = "mm2.png", width = "80%")
```

[11]:

$$\underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{\mathbf{X}} \quad \text{and} \quad \underbrace{\begin{pmatrix} e & f \\ g & h \end{pmatrix}}_{\mathbf{Y}}.$$

```
[12]: Image(filename = "mm3.png", width = "80%")
```

[12]:

$$\mathbf{X} \cdot \mathbf{Y} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}.$$

```
[14]: Image(filename = "mm4.png", width = "100%")
```

[14]:

Straightforward Matrix Multiplication

Input: $n \times n$ integer matrices $\mathbf{X}$ and $\mathbf{Y}$.
Output: $\mathbf{Z} = \mathbf{X} \cdot \mathbf{Y}$.

```
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        $\mathbf{Z}[i][j] := 0$
        for $k := 1$ to $n$ do
            $\mathbf{Z}[i][j] := \mathbf{Z}[i][j] + \mathbf{X}[i][k] \cdot \mathbf{Y}[k][j]$
return $\mathbf{Z}$
```

There are three nested for loops. This results in n^3 inner loop iterations (one for each choice of $i, j, k \in \{1, 2, \cdots, n\}$), and the algorithm performs a constant number of operations in each iteration (one multiplication and one addition).

The asymptotic running time of the straightforward algorithm for matrix multiplication is $\Theta(n^3)$

0.2 A Divide-and-Conquer Approach

- To apply the divide-and-conquer paradigm, we need to figure out how to divide the input into smaller subproblems and how to combine the solutions of these subproblems into a solution for the original problem.
- The simplest way to divide a square matrix into smaller square submatrices is to slice it in half, both vertically and horizontally.
- In other words, write

```
[15]: Image(filename = "mm5.png", width = "80%")
```

[15]:

$$\mathbf{X} = \begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{pmatrix} \quad \text{and} \quad \mathbf{Y} = \begin{pmatrix} \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} \end{pmatrix},$$

where A, B … and H are all n/2 X n/2 matrices.

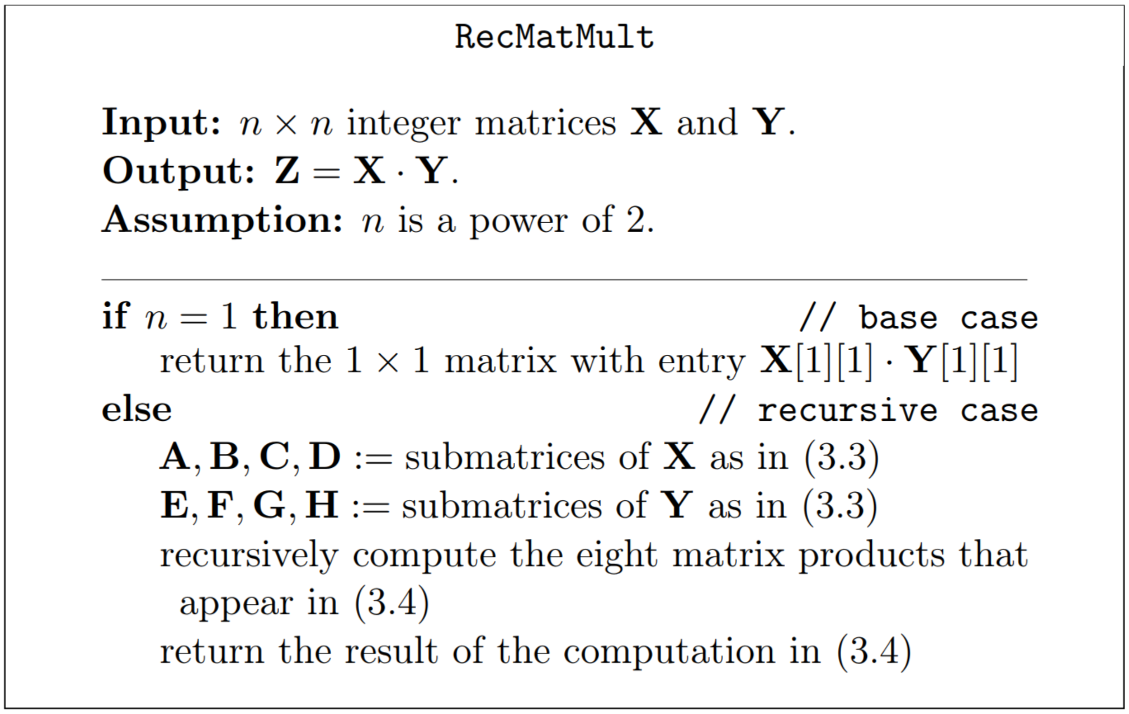
```
[16]: Image(filename = "mm6.png", width = "80%")
```

[16]:

$$\mathbf{X}\cdot\mathbf{Y} = \begin{pmatrix} \mathbf{A}\cdot\mathbf{E}+\mathbf{B}\cdot\mathbf{G} & \mathbf{A}\cdot\mathbf{F}+\mathbf{B}\cdot\mathbf{H} \\ \mathbf{C}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{G} & \mathbf{C}\cdot\mathbf{F}+\mathbf{D}\cdot\mathbf{H} \end{pmatrix},$$

```
[18]: Image(filename = "mm7.png", width = "100%")
```

[18]:



RecMatMult

Input: $n \times n$ integer matrices $\mathbf{X}$ and $\mathbf{Y}$.
Output: $\mathbf{Z} = \mathbf{X}\cdot\mathbf{Y}$.
Assumption: n is a power of 2.

```
if $n = 1$ then                                        // base case
    return the $1 \times 1$ matrix with entry $\mathbf{X}[1][1] \cdot \mathbf{Y}[1][1]$
else                                                   // recursive case
    $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ := submatrices of $\mathbf{X}$ as in (3.3)
    $\mathbf{E}, \mathbf{F}, \mathbf{G}, \mathbf{H}$ := submatrices of $\mathbf{Y}$ as in (3.3)
    recursively compute the eight matrix products that
      appear in (3.4)
    return the result of the computation in (3.4)
```

0.3 Strassen's algorithm

Let X and Y denote the two $n \times n$ input matrices, and define $A, B, \cdots, H$ as defined above.

The seven recursive matrix multiplications performed by Strassen's algorithm are:

```
[19]: Image(filename = "mm8.png", width = "45%")
```

[19]:

$$\begin{aligned}
\mathbf{P}_1 &= \mathbf{A}\cdot(\mathbf{F}-\mathbf{H})\\
\mathbf{P}_2 &= (\mathbf{A}+\mathbf{B})\cdot\mathbf{H}\\
\mathbf{P}_3 &= (\mathbf{C}+\mathbf{D})\cdot\mathbf{E}\\
\mathbf{P}_4 &= \mathbf{D}\cdot(\mathbf{G}-\mathbf{E})\\
\mathbf{P}_5 &= (\mathbf{A}+\mathbf{D})\cdot(\mathbf{E}+\mathbf{H})\\
\mathbf{P}_6 &= (\mathbf{B}-\mathbf{D})\cdot(\mathbf{G}+\mathbf{H})\\
\mathbf{P}_7 &= (\mathbf{A}-\mathbf{C})\cdot(\mathbf{E}+\mathbf{F}).
\end{aligned}$$

The final matrix multiplication result can be obtained using seven sub problem solutions $P_1, P_2, \cdots, P_7$ using

```
[20]: Image(filename = "mm9.png", width = "70%")
```

[20]:

$$\begin{aligned}
\mathbf{X}\cdot\mathbf{Y} &= \left(\begin{array}{c|c} \mathbf{A}\cdot\mathbf{E}+\mathbf{B}\cdot\mathbf{G} & \mathbf{A}\cdot\mathbf{F}+\mathbf{B}\cdot\mathbf{H} \\ \hline \mathbf{C}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{G} & \mathbf{C}\cdot\mathbf{F}+\mathbf{D}\cdot\mathbf{H} \end{array}\right)\\
&= \left(\begin{array}{c|c} \mathbf{P}_5+\mathbf{P}_4-\mathbf{P}_2+\mathbf{P}_6 & \mathbf{P}_1+\mathbf{P}_2 \\ \hline \mathbf{P}_3+\mathbf{P}_4 & \mathbf{P}_1+\mathbf{P}_5-\mathbf{P}_3-\mathbf{P}_7 \end{array}\right).
\end{aligned}$$

Let's check the equality of top left matrices

```
[21]: Image(filename = "mm10.png", width = "70%")
```

[21]:

$$\begin{aligned}\mathbf{P}_5+\mathbf{P}_4-\mathbf{P}_2+\mathbf{P}_6 = {} & (\mathbf{A}+\mathbf{D})\cdot(\mathbf{E}+\mathbf{H})+\mathbf{D}\cdot(\mathbf{G}-\mathbf{E}) \\ & \quad -(\mathbf{A}+\mathbf{B})\cdot\mathbf{H}+(\mathbf{B}-\mathbf{D})\cdot(\mathbf{G}+\mathbf{H}) \\ = {} & \mathbf{A}\cdot\mathbf{E}+\mathbf{A}\cdot\mathbf{H}+\mathbf{D}\cdot\mathbf{E}+\mathbf{D}\cdot\mathbf{H}+\mathbf{D}\cdot\mathbf{G} \\ & \quad -\mathbf{D}\cdot\mathbf{E}-\mathbf{A}\cdot\mathbf{H}-\mathbf{B}\cdot\mathbf{H}+\mathbf{B}\cdot\mathbf{G} \\ & \quad +\mathbf{B}\cdot\mathbf{H}-\mathbf{D}\cdot\mathbf{G}-\mathbf{D}\cdot\mathbf{H} \\ = {} & \mathbf{A}\cdot\mathbf{E}+\mathbf{B}\cdot\mathbf{G}.\end{aligned}$$

0.4 Example

[22]:
```
Image(filename = "strassen1.png", width = "70%")
```

[22]:

$$A = \begin{bmatrix} 0 & 0 & 3 & 4 \\ 5 & 6 & 0 & 0 \\ 0 & 0 & 6 & 5 \\ 4 & 3 & 0 & 0 \end{bmatrix}, \text{ and } B = \begin{bmatrix} 0 & 0 & 6 & 5 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$$

[23]:
```
Image(filename = "strassen2.png", width = "70%")
```

[23]:

$$\begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

[24]:
```
Image(filename = "strassen3.png", width = "70%")
```

[24]:

$$= \left(\begin{bmatrix}0&0\\5&6\end{bmatrix}+\begin{bmatrix}6&5\\0&0\end{bmatrix}\right)\times\left(\begin{bmatrix}0&0\\4&3\end{bmatrix}+\begin{bmatrix}3&4\\0&0\end{bmatrix}\right)$$

$$= \begin{bmatrix}6&5\\5&6\end{bmatrix}\times\begin{bmatrix}3&4\\4&3\end{bmatrix}=\begin{bmatrix}18+20&24+15\\15+24&20+18\end{bmatrix}=\begin{bmatrix}38&39\\39&38\end{bmatrix}$$

[25]:
```
Image(filename = "strassen3.png", width = "70%")
```
[25]:

$$= \left(\begin{bmatrix}0&0\\5&6\end{bmatrix}+\begin{bmatrix}6&5\\0&0\end{bmatrix}\right)\times\left(\begin{bmatrix}0&0\\4&3\end{bmatrix}+\begin{bmatrix}3&4\\0&0\end{bmatrix}\right)$$

$$= \begin{bmatrix}6&5\\5&6\end{bmatrix}\times\begin{bmatrix}3&4\\4&3\end{bmatrix}=\begin{bmatrix}18+20&24+15\\15+24&20+18\end{bmatrix}=\begin{bmatrix}38&39\\39&38\end{bmatrix}$$

[26]:
```
Image(filename = "strassen3.png", width = "70%")
```
[26]:

$$= \left(\begin{bmatrix}0&0\\5&6\end{bmatrix}+\begin{bmatrix}6&5\\0&0\end{bmatrix}\right)\times\left(\begin{bmatrix}0&0\\4&3\end{bmatrix}+\begin{bmatrix}3&4\\0&0\end{bmatrix}\right)$$

$$= \begin{bmatrix}6&5\\5&6\end{bmatrix}\times\begin{bmatrix}3&4\\4&3\end{bmatrix}=\begin{bmatrix}18+20&24+15\\15+24&20+18\end{bmatrix}=\begin{bmatrix}38&39\\39&38\end{bmatrix}$$

[ ]:
```
import numpy as np

def split(mat):
  r, c = mat.shape
r2, c2 = r //2, c//2
return mat[: r2,: cl2], mat[: r2, cl2: ], mat[r2: ,: c2], mat[r2: , c2: ]

def strassen(x, y):
  if len(x) == 1:
  return x * y
a, b, c, d = split(x)
```

```
e, f, g, h = split(y)

p1 = strassen(a, f - h)
p2 = strassen(a + b, h)
p3 = strassen(c + d, e)
p4 = strassen(d, g - e)
p5 = strassen(a + d, e + h)
p6 = strassen(b - d, g + h)
p7 = strassen(a - c, e + f)
z11 = p5 + p4 - p2 + p6
z12 = p1 + p2
z21 = p3 + p4
z22 = p1 + p5 - p3 - p7

z = np.vstack((np.hstack((z11, z12)), np.hstack((z21, z22))))

return z

x = [[0, 0 , 3, 4] [5, 6, 0, 0]]
result = strassen(x, y)
print(result)
```

0.5 Quick Sort Algorithm

- QuickSort was invented by Tony Hoare, in 1959, when he was just 25 years old. Hoare went on to make numerous fundamental contributions in programming languages and was awarded the ACM Turing Award---the equivalent of the Nobel Prize in computer science---in 1980.
- We already know one fast sorting algorithm (MergeSort)---why do we need another?
- On the practical side, QuickSort is competitive with and often superior to MergeSort, and for this reason is the default sorting method in many programming libraries.
- The big win for QuickSort over MergeSort is that it runs in place---it operates on the input array only through repeated swaps of pairs of elements, and for this reason needs to allocate only a minuscule amount of additional memory for intermediate computations.

0.6 Partitioning Around a Pivot

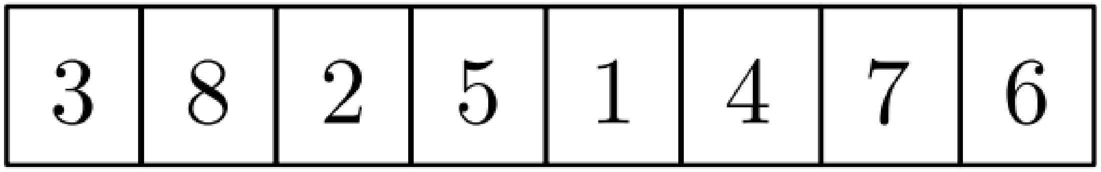
QuickSort is built around a fast subroutine for ``partial sorting,'' whose responsibility is to partition an array around a ``pivot element.''

1. Choose a pivot element. First, choose one element of the array to act as a pivot element. For now just use the first element of the array (above, the ``3'').

2. Rearrange the input array around the pivot. Given the pivot element p, the next task is to arrange the elements of the array so that everything before p in the array is less than p, and everything after p is greater than p. For example, with the input array,

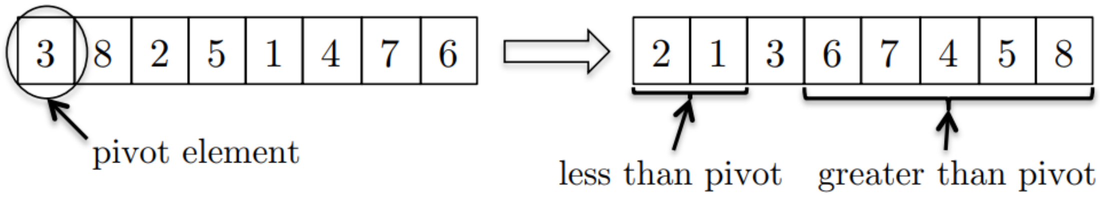
[27]:
```
Image(filename = "qs.png", width = "35%")
```

[27]:



[28]:
```
Image(filename = "qs1.png", width = "80%")
```

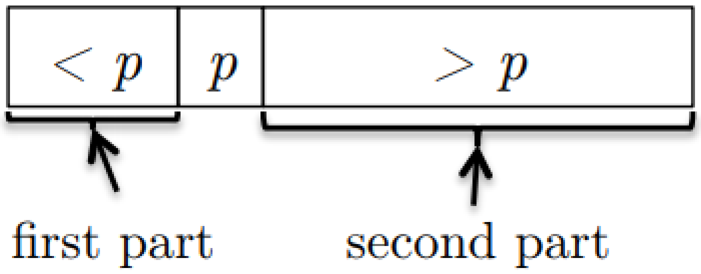
[28]:



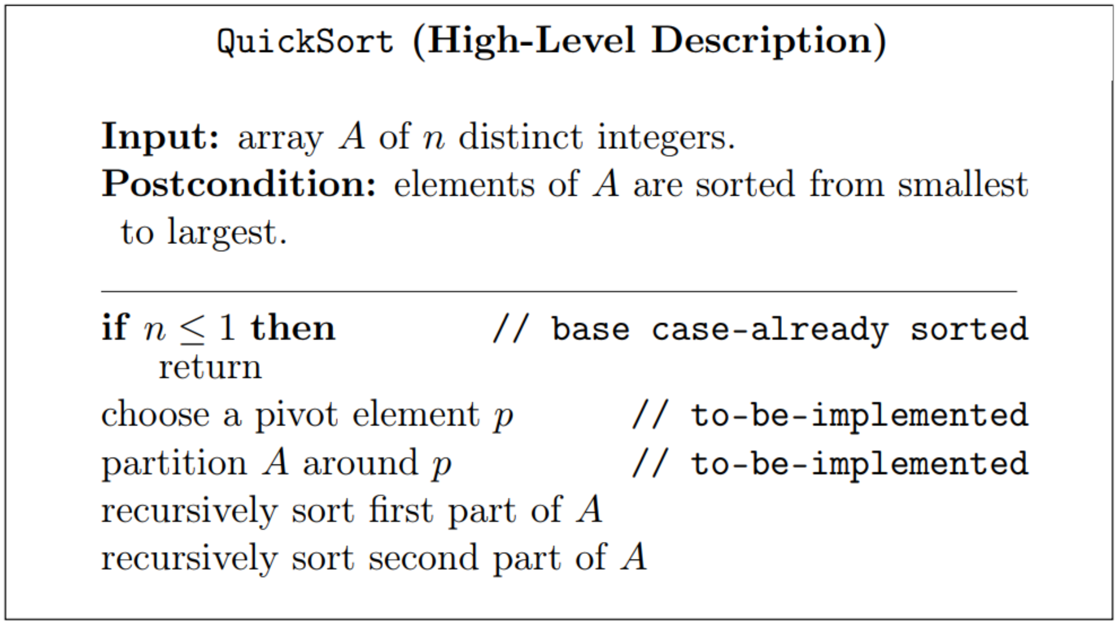
The two key facts about this partition subroutine are.

- **Fast**. The partition subroutine has a blazingly fast implementation, running in linear (O(n)) time. Even better, and key to the practical utility of QuickSort, the subroutine can be implemented in place, with next to no memory beyond that occupied by the input array.
- **Significant progress**. Partitioning an array around a pivot element makes progress toward sorting the array.
  - First, the pivot element winds up in its rightful position, meaning the same position as in the sorted version of the input array (with all smaller elements before it and all larger elements after it).
  - Second, partitioning reduces the sorting problem to two smaller sorting problems: sorting the elements less than the pivot (which conveniently occupy their own subarray) and the elements greater than the pivot (also in their own subarray). After recursively sorting the elements in each of these two subarrays, the algorithm is done.
  - One of the subproblems might be empty, if the minimum or maximum element is chosen as the pivot. In this case, the corresponding recursive call can be skipped.

[30]:
```
Image(filename = "qs1b.png", width = "35%")
```
[30]:



[32]:
```
Image(filename = "qs2.png", width = "100%")
```
[32]:



QuickSort **(High-Level Description)**

Input: array A of n distinct integers.
Postcondition: elements of A are sorted from smallest to largest.

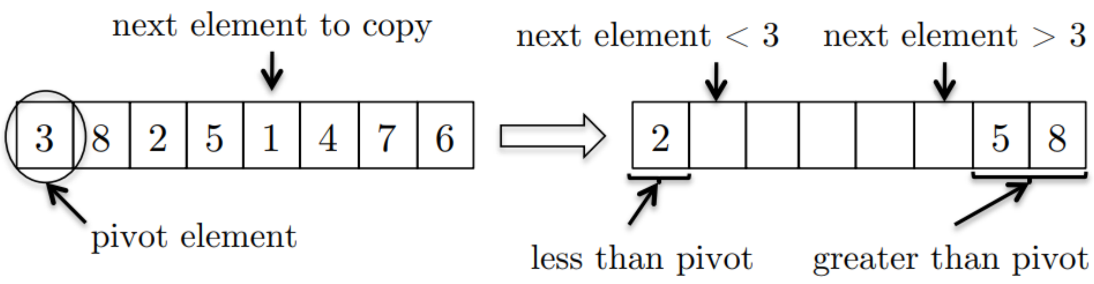
```
if $n \leq 1$ then              // base case-already sorted
    return
choose a pivot element $p$      // to-be-implemented
partition $A$ around $p$        // to-be-implemented
recursively sort first part of $A$
recursively sort second part of $A$
```

0.7 Partitioning Around a Pivot Element

- It's easy to come up with a linear-time partitioning subroutine if we don't care about allocating additional memory.
- One approach is to do a single scan over the input array A and copy over its non-pivot elements one by one into a new array B of the same length, populating B both from its front (for elements less than p) and its back (for elements bigger than p). The pivot element can be copied into the remaining entry of B after all the non-pivot elements have been processed.

```
[34]: Image(filename = "qs3.png", width = "80%")
```

[34]:

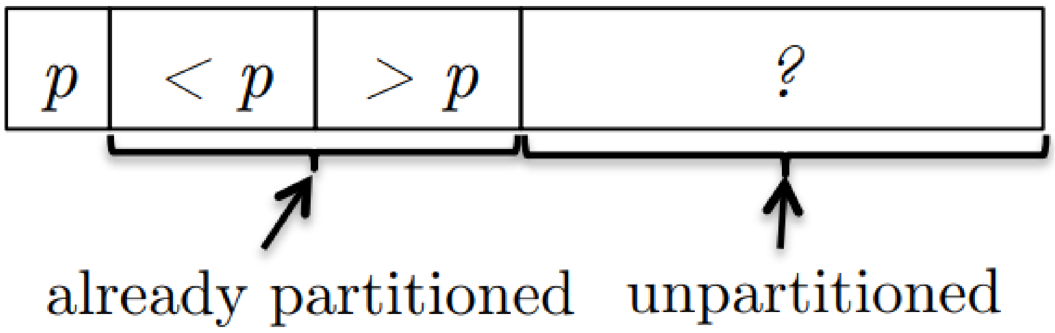


- Since this subroutine does only O(1) work for each of the n elements in the input array, its running time is O(n).

How do we partition an array around a pivot element while allocating almost no additional memory?

```
[35]: Image(filename = "qs4.png", width = "80%")
```

[35]:

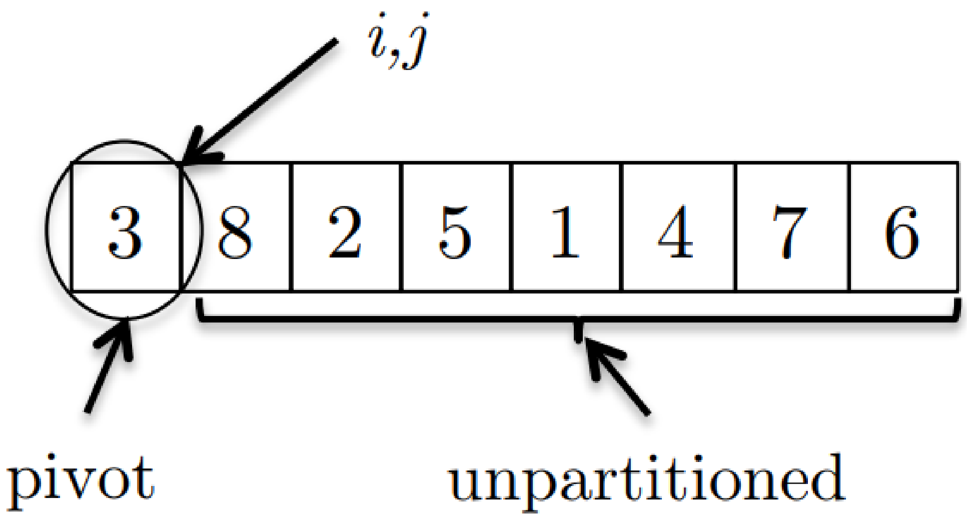


0.7.1 In-Place Implementation

- Let the first element be the pivot element.
- Let us use index i, to keep track of the boundary between the elements less than the pivot and those greater than the pivot.
- Let us use index j, to keep track of the boundary between the non-pivot elements we've already looked at and those we haven't processed.
- Invariant: all elements between the pivot and i are less than the pivot, and all elements between i and j are greater than the pivot. Both i and j are initialized to the boundary between the pivot element and the rest.

```
[36]: Image(filename = "qs5.png", width = "80%")
```

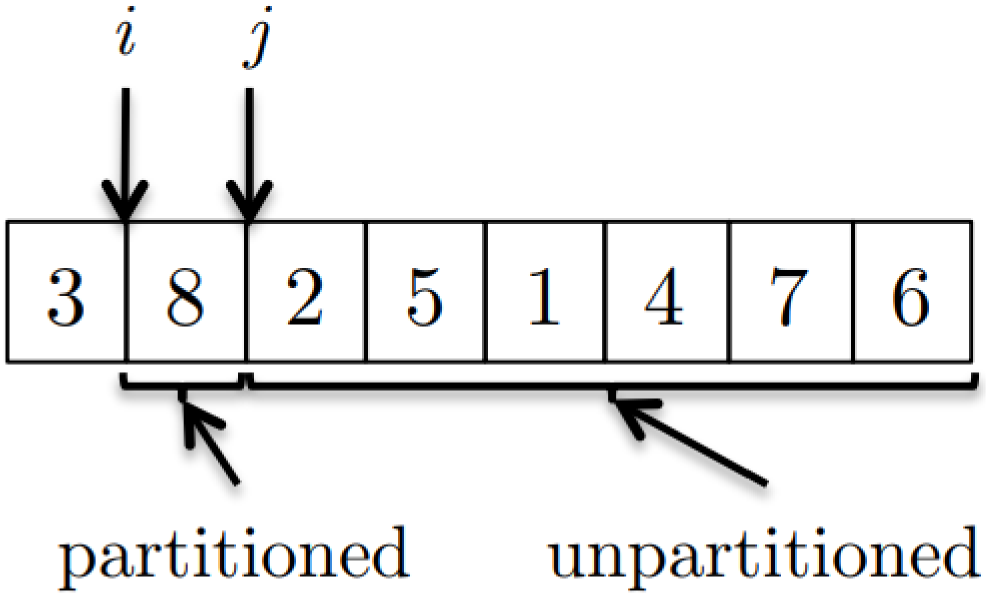
[36]:



- Each iteration, the subroutine looks at one new element, and increments j. Additional work may or may not be required to maintain the invariant. The first time we increment j in our example, we get:

[37]:
```
Image(filename = "qs6.png", width = "80%")
```

[37]:

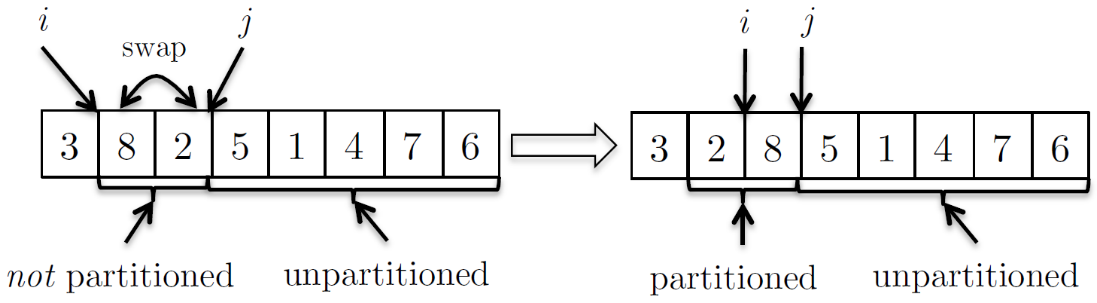


- There are no elements between the pivot and i, and the only element between i and j (the ``8'') is greater than the pivot, so the invariant still holds.

- After incrementing j a second time, there is an element between i and j that is less than the pivot (the ``2''), a violation of the invariant.
- To restore the invariant, we swap the ``8'' with the ``2,'' and also increment i, so that it is wedged between the ``2'' and the ``8'' and again delineates the boundary between processed elements less than and greater than the pivot:

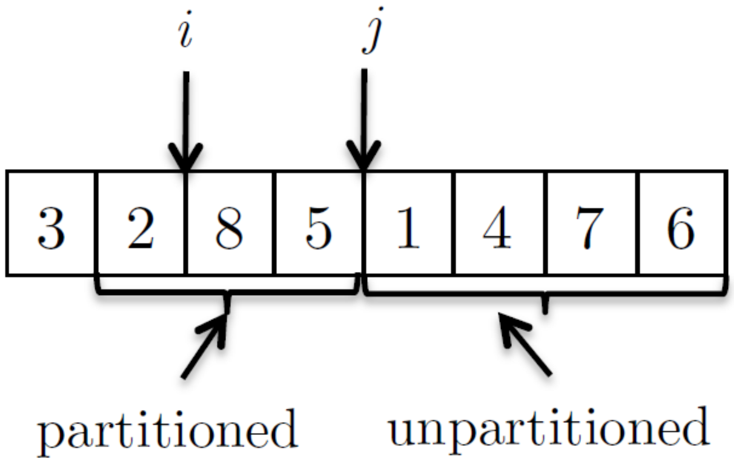
[38]:
```
Image(filename = "qs7.png", width = "100%")
```

[38]:



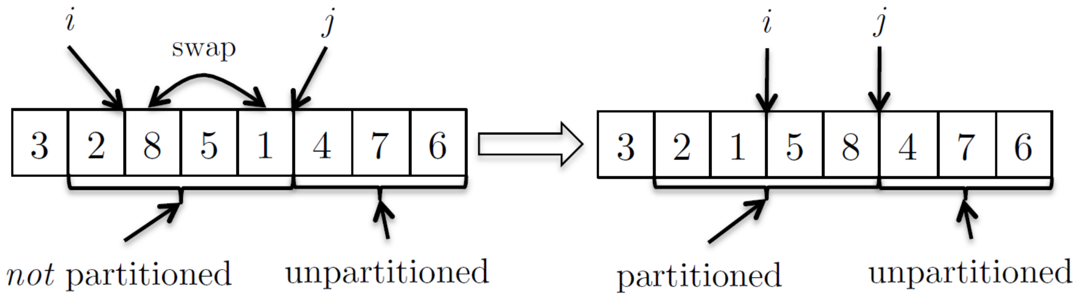
[39]:
```
Image(filename = "qs8.png", width = "75%")
```

[39]:



[40]:
```
Image(filename = "qs9.png", width = "75%")
```

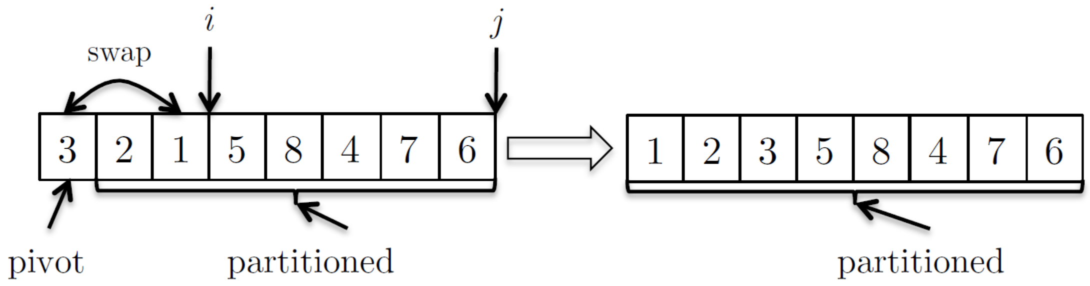
[40]:



- After all the elements have been processed and everything after the pivot has been partitioned, we conclude with the final swap of the pivot element and the last element smaller than it:

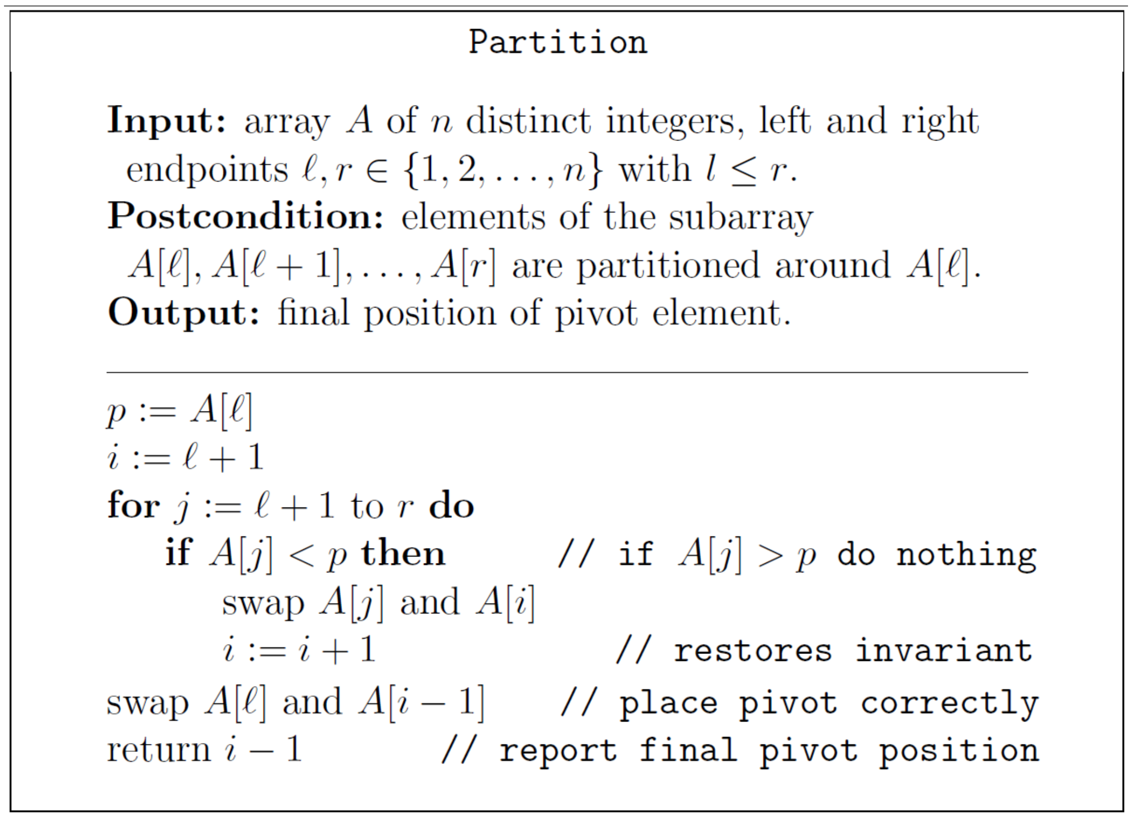
```
[41]: Image(filename = "qs10.png", width = "75%")
```

[41]:



```
[44]: Image(filename = "qs11.png", width = "100%")
```

[44]:



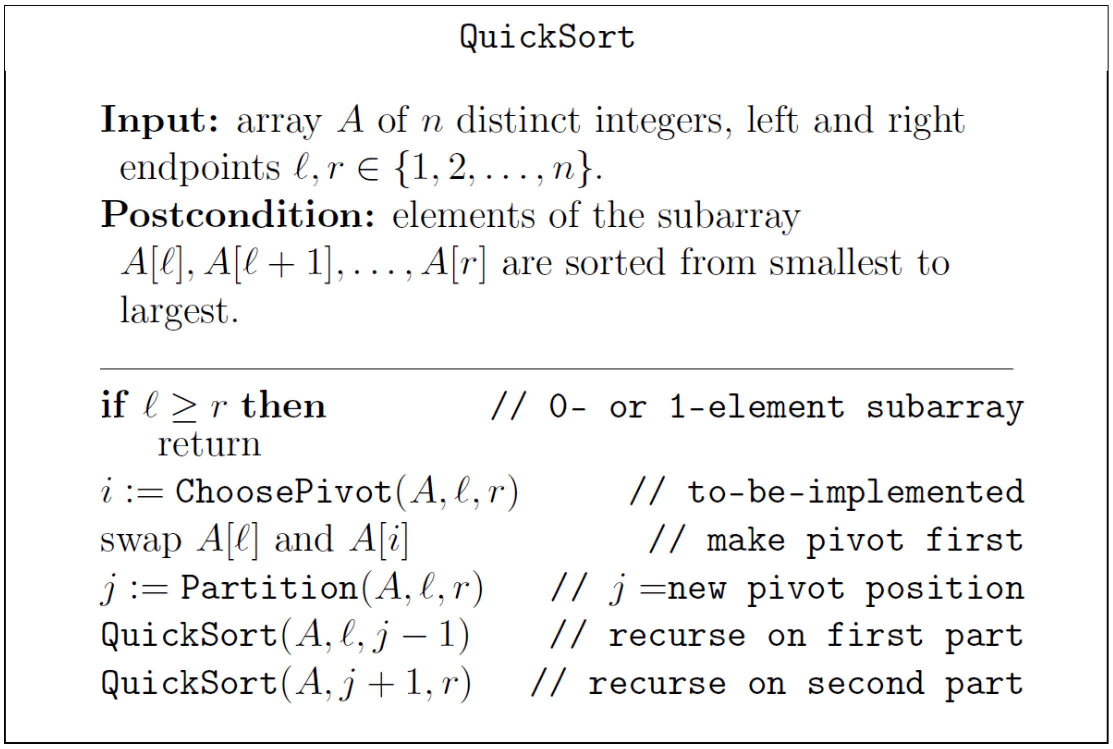
Partition

Input: array A of n distinct integers, left and right endpoints $\ell, r \in \{1, 2, \ldots, n\}$ with $l \leq r$.
Postcondition: elements of the subarray $A[\ell], A[\ell+1], \ldots, A[r]$ are partitioned around $A[\ell]$.
Output: final position of pivot element.

```
$p := A[\ell]$
$i := \ell + 1$
for $j := \ell + 1$ to $r$ do
    if $A[j] < p$ then          // if $A[j] > p$ do nothing
        swap $A[j]$ and $A[i]$
        $i := i + 1$                // restores invariant
swap $A[\ell]$ and $A[i-1]$      // place pivot correctly
return $i - 1$            // report final pivot position
```

```
[45]: Image(filename = "qs12.png", width = "100%")
```

```
[45]:
```



QuickSort

Input: array A of n distinct integers, left and right endpoints $\ell, r \in \{1, 2, \ldots, n\}$.
Postcondition: elements of the subarray $A[\ell], A[\ell+1], \ldots, A[r]$ are sorted from smallest to largest.

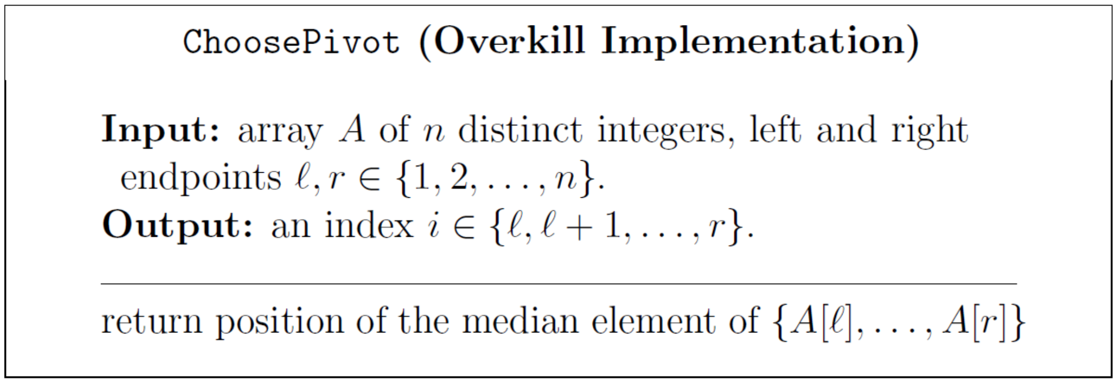
```
if $\ell \geq r$ then                     // 0- or 1-element subarray
    return
$i$ := ChoosePivot($A, \ell, r$)          // to-be-implemented
swap $A[\ell]$ and $A[i]$                 // make pivot first
$j$ := Partition($A, \ell, r$)            // $j$ =new pivot position
QuickSort($A, \ell, j-1$)                 // recurse on first part
QuickSort($A, j+1, r$)                    // recurse on second part
```

0.8 Choice of pivot element

- For QuickSort to be quick, it's important that ``good'' pivot elements are chosen, meaning pivot elements that result in two subproblems of roughly the same size.
- The most perfectly balanced split is achieved by the median element of the array, meaning the element for which the same number of other elements are less than it and greater than it.

```
[46]: Image(filename = "qs13.png", width = "75%")
```

[46]:



ChoosePivot (Overkill Implementation)

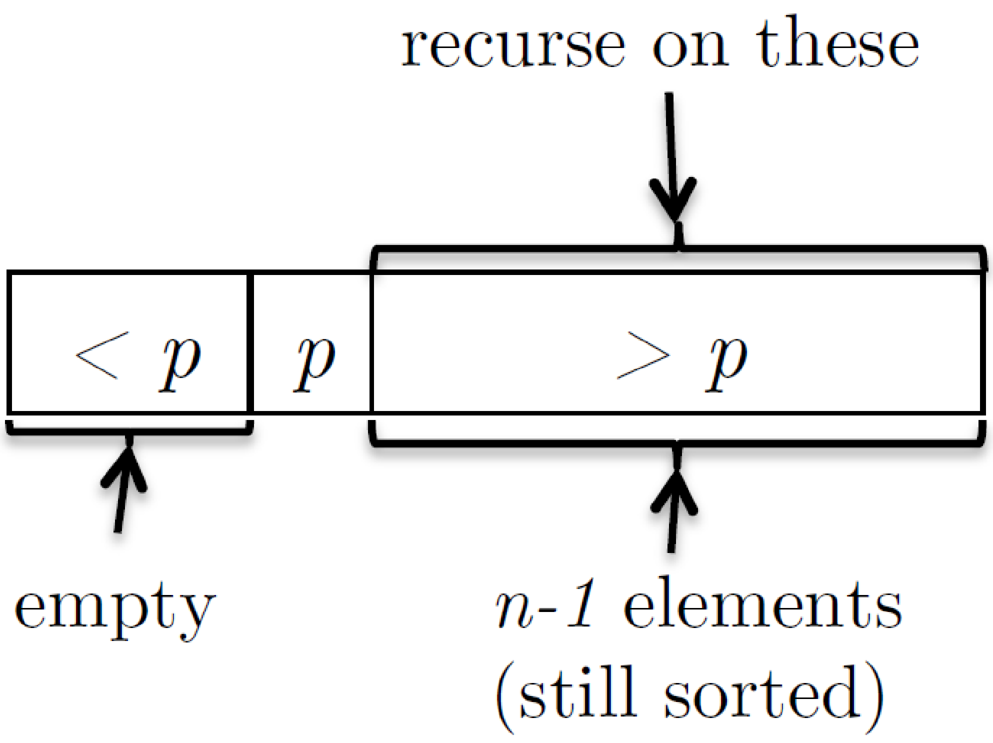
Input: array A of n distinct integers, left and right endpoints $\ell, r \in \{1, 2, \ldots, n\}$.
Output: an index $i \in \{\ell, \ell+1, \ldots, r\}$.

```
return position of the median element of $\{A[\ell], \ldots, A[r]\}$
```

- The combination of naively chosen pivots and an already-sorted input array causes QuickSort to run in $\Theta(n^2)$ time, which is much worse than MergeSort and no better than simple algorithms such as InsertionSort. What goes wrong? The Partition subroutine in the outermost call to QuickSort, with the first (smallest) element as the pivot, does nothing: it sweeps over the array, and since it only encounters elements greater than the pivot, it never swaps any pair of elements. After this call to Partition completes, the picture is:

[47]:
```
Image(filename = "qs14.png", width = "55%")
```

[47]:



- In the non-empty recursive call, the pattern recurs: the subarray is already sorted, the first (smallest) element is chosen as the pivot, and there is one empty recursive call and one recursive call that is passed a subarray of n - 2 elements. And so on.

- In the end, the Partition subroutine is invoked on subarrays of length n, n - 1, n - 2, . . . , 2. Since the work done in one call to Partition is proportional to the length of the call's subarray, the total amount of work done by QuickSort in this case is proportional to

[48]:
```
Image(filename = "qs15.png", width = "55%")
```

[48]:

$$\underbrace{n + (n-1) + (n-2) + \cdots + 1}_{=\Theta(n^2)}$$

0.9 Quick sort asymptotic analysis

- In the best-case scenario, QuickSort runs in $\Theta(n \log n)$ time.
- The reason is that its running time is governed by the exact same recurrence that governs the running time of MergeSort. That is, if T(n) denotes the running time of this implementation of QuickSort on arrays of length n, then

[49]:
```
Image(filename = "qs16.png", width = "55%")
```

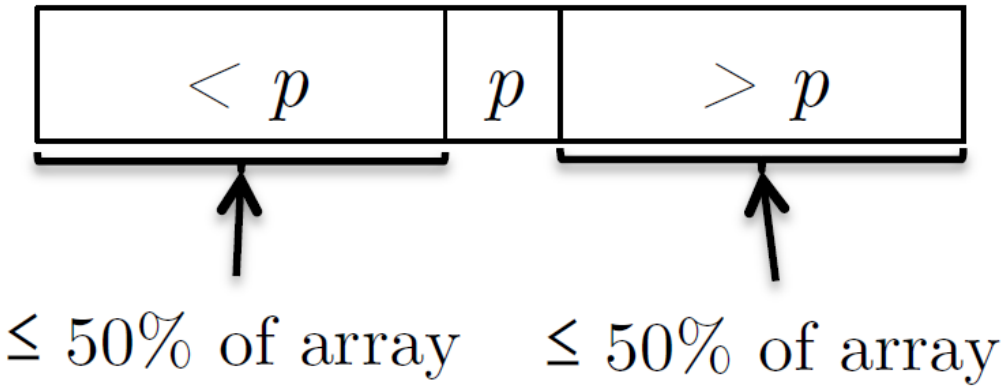
[49]:

$$T(n) = \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{since pivot = median}} + \underbrace{\Theta(n)}_{\texttt{ChoosePivot \& Partition}} .$$

- The primary work done by a call to QuickSort outside its recursive calls occurs in its ChoosePivot and Partition subroutines. We're assuming that the former is $\Theta(n)$, and in the previous we proved that the latter is also $\Theta(n)$.
- Since we're using the median element as the pivot element, we get a perfect split of the input array and each recursive call gets a subarray with at most $n/2$ elements:

[50]:
```
Image(filename = "qs17.png", width = "55%")
```

[50]:



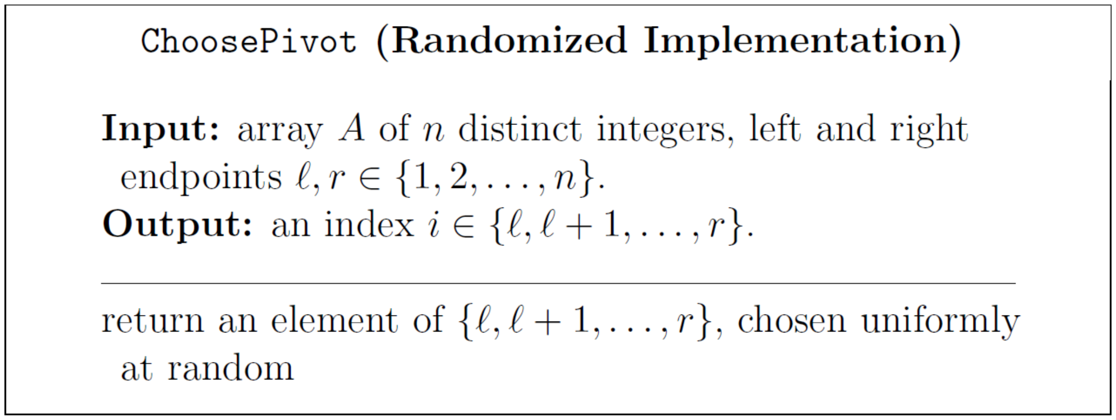
- Applying the master method with a = b = 2 and d = 1 then gives $T(n) = \Theta(n \log n)$.

0.10 Randomized Quick Sort

- The simplest way to incorporate randomness into QuickSort, which turns out to be extremely effective, is to always choose pivot elements uniformly at random.

[52]:
```
Image(filename = "qs18.png", width = "75%")
```

[52]:



ChoosePivot (**Randomized Implementation**)

Input: array A of n distinct integers, left and right endpoints $\ell, r \in \{1, 2, \ldots, n\}$.
Output: an index $i \in \{\ell, \ell + 1, \ldots, r\}$.

return an element of $\{\ell, \ell + 1, \ldots, r\}$, chosen uniformly at random

- The running time of randomized QuickSort, with pivot elements chosen at random, is not always the same.
- The algorithm's running time fluctuates between $\Theta(n \log n)$ and $\Theta(n^2)$
- Which occurs more frequently, the best-case scenario or the worst-case scenario? Amazingly, the performance of QuickSort is almost always close

to its best-case performance.

- For every input array of length $n \geq 1$, the average running time of randomized QuickSort is $O(n \log n)$.