# DECOMPOSITION TECHNIQUES

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

PURDUE NExT

# RECURSIVE DECOMPOSITION

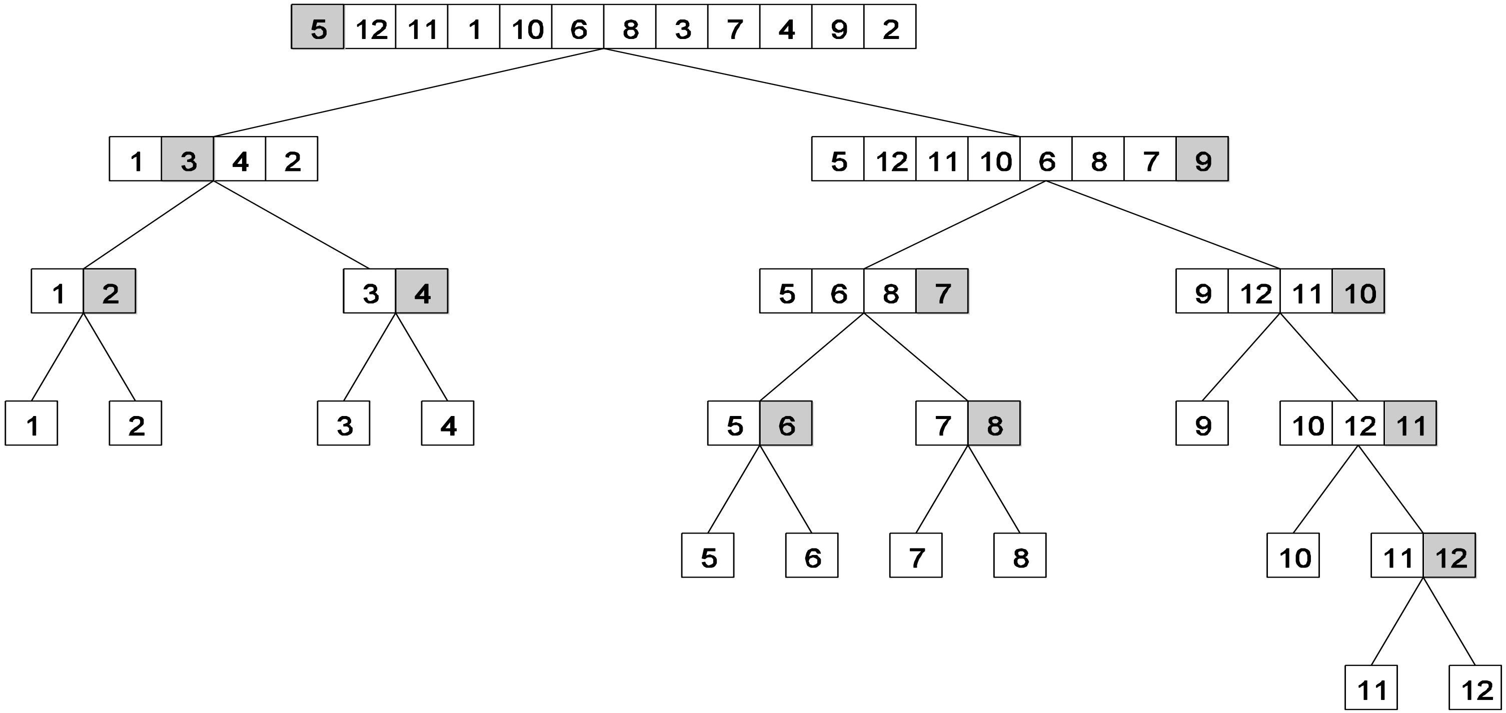
Generally suited to problems that are solved using the divide-and-conquer strategy.

A given problem is first decomposed into a set of sub-problems.

These sub-problems are recursively decomposed further until a desired granularity is reached.

# RECURSIVE DECOMPOSITION: EXAMPLE

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.




In this example, once the list has been partitioned around the pivot, each sub-list can be processed concurrently (i.e., each sub-list represents an independent subtask). This can be repeated recursively.

# RECURSIVE DECOMPOSITION: EXAMPLE

The problem of finding the minimum number in a given list (or indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm. The following algorithm illustrates this.

We first start with a simple serial loop for computing the minimum entry in a given list:

```
procedure SERIAL_MIN (A, n)
begin
min = A[0];
for i := 1 to n − 1 do
        if (A[i] < min) min := A[i];
endfor;
return min;
end SERIAL_MIN
```

PURDUE NExT

# RECURSIVE DECOMPOSITION: EXAMPLE

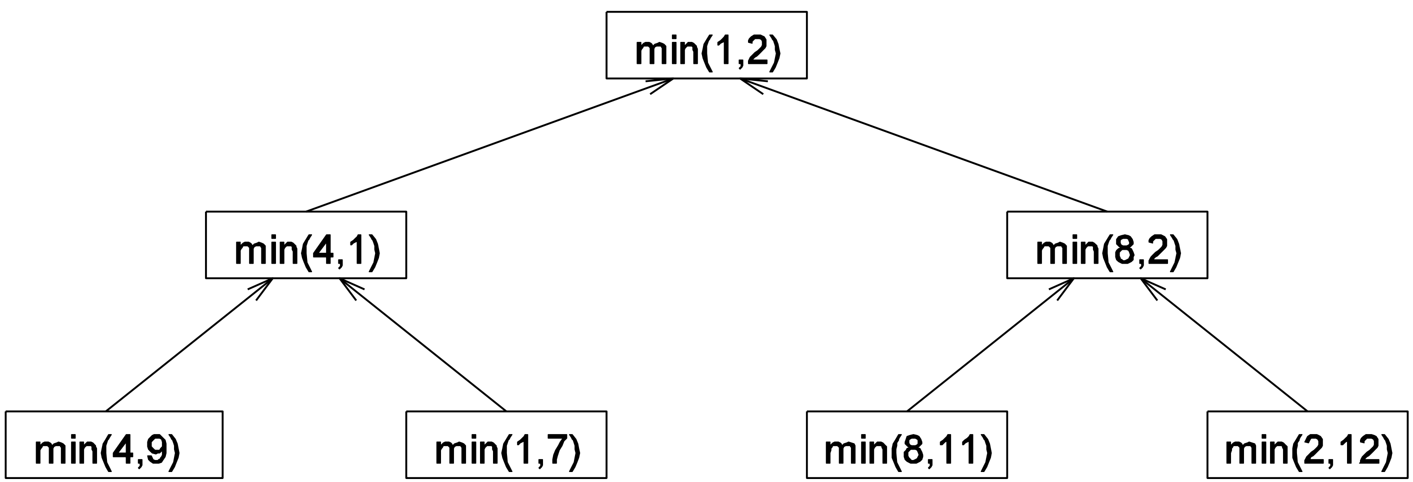
We can rewrite the loop as follows:

```
procedure RECURSIVE_MIN (A, n)
begin
if ( n = 1 ) then
    min := A [0]  ;
else
    lmin := RECURSIVE_MIN ( A, n/2 );
    rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
    if (lmin  < rmin) then
            min := lmin;
    else
            min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

PURDUE NExT

# RECURSIVE DECOMPOSITION: EXAMPLE

The code in the previous foil can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of finding the minimum number in the set {4, 9, 1, 7, 8, 11, 2, 12}. The task dependency graph associated with this computation is as follows:



# DATA DECOMPOSITION

Identify the data on which computations are performed.

Partition this data across various tasks.

This partitioning induces a decomposition of the problem.

Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.

# DATA DECOMPOSITION: OUTPUT DATA DECOMPOSITION

Often, each element of the output can be computed independently of others (but simply as a function of the input).

A partition of the output across tasks decomposes the problem naturally.

# OUTPUT DATA DECOMPOSITION: EXAMPLE

Consider the problem of multiplying two ***n*** x ***n*** matrices ***A*** and ***B*** to yield matrix ***C***. The output matrix ***C*** can be partitioned into four tasks as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

PURDUE NExT

# OUTPUT DATA DECOMPOSITION: EXAMPLE

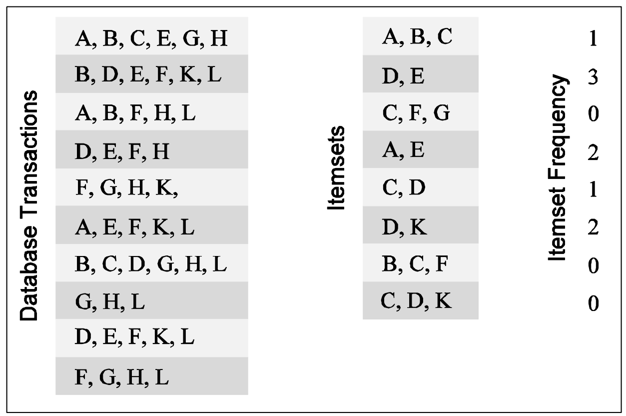
A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previus foil, with identical output data distribution, we can derive the following two (other) decompositions:

| **Decomposition I** | **Decomposition II** |
|---|---|
| Task 1: $\boldsymbol{C_{1,1} = A_{1,1} B_{1,1}}$ | Task 1: $\boldsymbol{C_{1,1} = A_{1,1} B_{1,1}}$ |
| Task 2: $\boldsymbol{C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}}$ | Task 2: $\boldsymbol{C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}}$ |
| Task 3: $\boldsymbol{C_{1,2} = A_{1,1} B_{1,2}}$ | Task 3: $\boldsymbol{C_{1,2} = A_{1,2} B_{2,2}}$ |
| Task 4: $\boldsymbol{C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}}$ | Task 4: $\boldsymbol{C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}}$ |
| Task 5: $\boldsymbol{C_{2,1} = A_{2,1} B_{1,1}}$ | Task 5: $\boldsymbol{C_{2,1} = A_{2,2} B_{2,1}}$ |
| Task 6: $\boldsymbol{C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}}$ | Task 6: $\boldsymbol{C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}}$ |
| Task 7: $\boldsymbol{C_{2,2} = A_{2,1} B_{1,2}}$ | Task 7: $\boldsymbol{C_{2,2} = A_{2,1} B_{1,2}}$ |
| Task 8: $\boldsymbol{C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}}$ | Task 8: $\boldsymbol{C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}}$ |

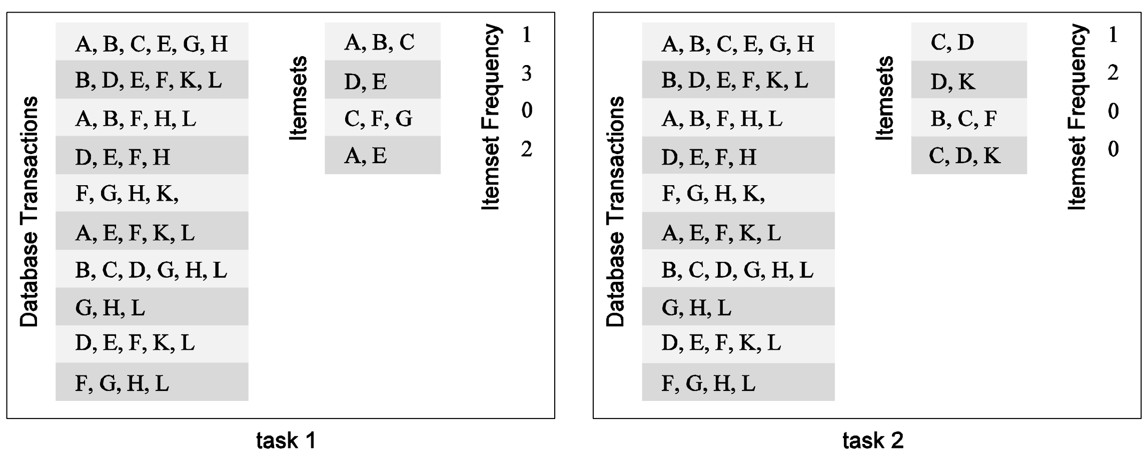
# OUTPUT DATA DECOMPOSITION: EXAMPLE

Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.

**(a) Transactions (input), itemsets (input), and frequencies (output)**




**(b) Partitioning the frequencies (and itemsets) among the tasks**



# OUTPUT DATA DECOMPOSITION: EXAMPLE

From the previous example, the following observations can be made:

If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.

If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

# INPUT DATA PARTITIONING

Generally applicable if each output can be naturally computed as a function of the input.

In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).

A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.

# INPUT DATA PARTITIONING: EXAMPLE

In the database counting example, the input (i.e., the transaction set) can be partitioned. This induces a task decomposition in which each task generates partial counts for all item sets. These are combined subsequently for aggregate counts.

**Partitioning the transactions among the tasks**

**task 1**

| Database Transactions | Itemsets | Itemset Frequency |
|---|---|---|
| A, B, C, E, G, H | A, B, C | 1 |
| B, D, E, F, K, L | D, E | 2 |
| A, B, F, H, L | C, F, G | 0 |
| D, E, F, H | A, E | 1 |
| F, G, H, K, | C, D | 0 |
| | D, K | 1 |
| | B, C, F | 0 |
| | C, D, K | 0 |

**task 2**

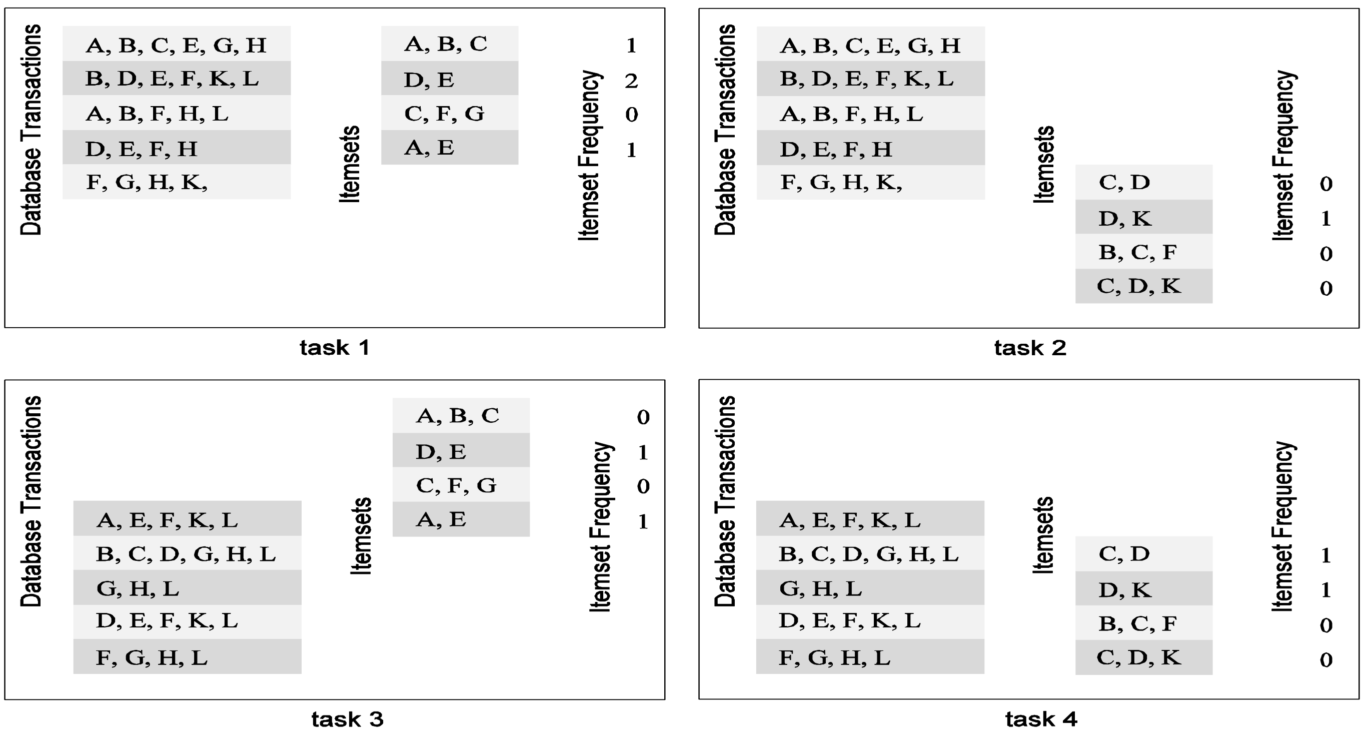
| Database Transactions | Itemsets | Itemset Frequency |
|---|---|---|
| | A, B, C | 0 |
| | D, E | 1 |
| | C, F, G | 0 |
| A, E, F, K, L | A, E | 1 |
| B, C, D, G, H, L | C, D | 1 |
| G, H, L | D, K | 1 |
| D, E, F, K, L | B, C, F | 0 |
| F, G, H, L | C, D, K | 0 |

PURDUE NExT

# PARTITIONING INPUT *AND* OUTPUT DATA

Often input and output data decomposition can be combined for a higher degree of concurrency. For the item set counting example, the transaction set (input) and itemset counts (output) can both be decomposed as follows:

**Partitioning both transactions and frequencies among the tasks**




PURDUE NExT