```
csharp
Copy Code
// Original
int Add(int x, int y) => x + y;
int result = Add(2, 3);

// After Inlining
int result = 2 + 3; // Directly replaces the method call
```

```
csharp
Copy Code
// Original
int result = 2 * 3 + 4;

// After Constant Folding
int result = 6 + 4; // Evaluated at compile time
```

```
csharp
Copy Code
// Original
if (false) {
    Console.WriteLine("This will never execute.");
}
```

```
// After Dead Code Elimination
// The entire block is removed
```

```
csharp
Copy Code
// Original
for (int i = 0; i < 4; i++) {
    Console.WriteLine(i);
}

// After Loop Unrolling
Console.WriteLine(0);
Console.WriteLine(1);
Console.WriteLine(2);
Console.WriteLine(3);
```

```
csharp
Copy Code
// Original
int a = 2 * 3;
int b = 2 * 3 + 1;

// After Common Subexpression Elimination
int temp = 2 * 3;
int a = temp;
```

```
int b = temp + 1;
```

```
csharp
Copy Code
// Original
int a = 5;
int b = 10;
int c = a + b;

// After Register Allocation
// a, b, and c might be stored in CPU registers instead of
memory
```

```
csharp
Copy Code
// Original
if (condition) {
    // Code path A
} else {
    // Code path B
}

// After Optimization
```

Reorganizes code to favor the most likely path based on profiling

```
csharp
Copy Code
// Original
void Method() {
    MyClass obj = new MyClass(); // Allocated on heap
}

// After Escape Analysis
// If obj does not escape the method, it could be allocated on the stack
```

```
csharp
Copy Code
// Original
void Print(object obj) {
    Console.WriteLine(obj.ToString());
}

// After Type Specialization
// If obj is known to be a string at runtime, it can optimize the call
```

```
csharp
Copy Code
// Original
int Factorial(int n) {
    if (n == 0) return 1;
    return n * Factorial(n - 1);
}

// After Tail Call Optimization
// The recursive call can be optimized to avoid increasing the call stack
```

```
csharp
Copy Code
// Original
for (int i = 0; i < 10; i++) {
    int x = 2; // Invariant
    Console.WriteLine(x + i);
}

// After Loop Invariant Code Motion
int x = 2; // Moved outside the loop
for (int i = 0; i < 10; i++) {
```

```
    Console.WriteLine(x + i);
}
```

```
csharp
Copy Code
class Base {
    virtual void Display() { Console.WriteLine("Base"); }
}

class Derived : Base {
    override void Display() { Console.WriteLine("Derived"); }
}

// After Inlining
// If the type is known at runtime, the call can be inlined
```

```
csharp
Copy Code
// Original
void ProcessData() {
    // Code that is executed frequently
}

// After PGO
```

The JIT optimizes frequently executed paths based on runtime profiling

```
csharp
Copy Code
// Original
for (int i = 0; i < 10; i++) {
    int x = ComputeValue(); // Computed every iteration
}

// After Code Hoisting
int x = ComputeValue(); // Computed once
for (int i = 0; i < 10; i++) {
    // Use x
}
```

```
csharp
Copy Code
// Original
for (int i = 0; i < 4; i++) {
    array[i] = array[i] * 2;
}

// After Vectorization
```

```
// The JIT may use SIMD instructions to process multiple elements at once
```

```
csharp
Copy Code
// Original
for (int i = 0; i < array.Length; i++) {
    Console.Write
```