Optimization

Logic Techniques

- Stop testing when answer is found (break out of loops as soon as possible)

```
negFound = false;
for (int i = 0; i < count; i++) {
        if (input[i] < 0) {
                negFound = true;
                break;
        }
}
```

- Order tests by frequency in switch and if-else structures

```
if ((c == '+') || (c == '-'))
        processMath(c);
else if ((c >= '0') && (c <= '9'))
        processDigit(c);
else if ((c >= 'a') && (c <= 'z'))
        processLetter(c);
```

  - Rearrange these tests based on how common different inputs are
- Substitute switch statement for if-else construct, or vice-versa
  - In Java, an if-else construct is about 6 times faster than a switch
  - But in Visual Basic, its 4 times slower
- Substitute table lookups for complicated expressions
  - Can be implemented using complicated logic

```
if ((a && !c) || (a && b && c))
        category = 1;
else if (b && !a) || (a && c && !b)
        category = 2;
else if (c && !a && !b)
        category = 3;
else
        category = 0;
```

  - But is faster with a lookup table:

```
// Define category table
static int categoryTable[2][2][2] = {
        // !b!c    !bc     b!c     bc
                0,      3,      2,      2,      // !a
                1,      2,      1,      1,      // a
};
...
category = categoryTable[a][b][c];
```

  - Use lazy evaluation
    - Eg. A 5000-entry table could be generated when the program starts
      - But if only a few entries are ever used, may be better to compute values as needed and then store them in the table
      - Cache them for further use

Loop Techniques

Unswitching:

- Switching is where a decision is made inside a loop on every iteration

```
for (i = 0; i < count; i++) {
        if (type == NET) { netSum += amount[i]; }
        else { grossSum += amount[i]; }
}
```

- The if construct should be held outside of the loop as `type` never changes
  - Note this likely requires that two loops must be maintained in parallel

Jamming (fusion):

- Combines two or more loops into one: their loop counters should be similar

```
for (i = 0; i < length; i++)
        employeeSalary[i] = 0.0;
for (i = 0; i < length; i++)
        employeeCode[i] = 'C';
// Better as
for (i = 0; i < length; i++) {
        employeeSalary[i] = 0.0;
        employeeCode[i] = 'C';
}
```

Unrolling:

- A *complete unrolling* replaces a loop with straight-line code
  - Practical only for short loops

```
for (i = 0; i < 10; i++) { a[i] = i; }

// Better as
a[0] = 0;
...
a[9] = 9;
```

- With *partial unrolling*, two or more cases are handled inside the loop instead of just one
  - The above example unrolled once becomes:

  ```
  for (i = 0; i < count - 1; i += 2) {
          a[i] = i;
          a[i + 1] = i + 1;
  }
  if (i == count - 1)
          a[count - 1] = count - 1;
  ```

  - Unrolled twice becomes

  ```
  for (i = 0; i < count - 2; i += 3) {
          a[i] = i;
          a[i + 1] = i + 1;
          a[i + 2] = i + 2;
  }
  if (i == count - 2) {
          a[count - 2] = count - 2;
          a[count - 1] = count - 1;
  }
  if (i == count - 1)
          a[count - 1] = count - 1;
  ```

Minimizing work inside loops:

- Put calculations that result in a constant before the loop

```
for (i = 0; i < rateCount; i++)
        netRate[i] = baseRate[i] * rates.discount() / 0.93;
// Better as
quantityDiscount = rates.discount() / 0.93;
for (i = 0; i < rateCount; i++)
        netRate[i] = baseRate[i] * quantityDiscount;
```

Sentinel Values:

- Are used to simplify loop control
  - Replaces expensive compound tests
- A sentinel is a special value that marks the end of an array
  - Is guaranteed to terminate a search through the loop
  - Declare the array one element bigger so it can hold the sentinel

```
found = FALSE;
i = 0;
while (!found && (i < count)) {
        if (item[i] == searchKey)
                found = TRUE;
        else
                i++;
}
if (found) { ... }
```

- With a sentinel, becomes:

```
item[count] = searchKey;
i = 0;
while (item[i] != searchKey)
        i++;

if (i < count) { /* Item is found */ }
```

Putting the busiest loop on the inside:

```
for (column = 0; column < 100; column++) {
        for (row = 0; row < 5; row++) {
                sum += table[row][column];
        }
}
```

- Loop operations: (Outer = 100) + (Inner = 100 * 5) = 600
- Switching the inner and outer loops end up with: (Outer = 5) + (Inner = 100 * 5) = 505

Strength Reduction:

- Replace an expensive operation with a cheaper operation
  - Eg. Replace multiplication with addition

```
for (i = 0; i < saleCount; i++)
        commission[i] = (i + 1) * revenue * baseCommission * discount;
```

- After strength reduction:

```
increment = revenue * baseCommission * discount;
cum = increment;
for (i = 0; i < saleCount; i++) {
        cum += increment;
        commission[i] = cum;
}
```

Routines

- Rewrite routines inline
  - C++ has the `inline` keyword
  - With other languages, use macros

```
#define SQUARE(x) ((x) * (x))
...
int a = 5, b;
b = SQUARE(a);
```

- Rewrite expensive system routines
  - Eg. `double log2(double x)` may give more precision than you need
    - Rounding integer version:

```
unsigned int log2(unsigned int x) {
        if (x < 2) return 0;
        if (x < 4) return 1;
        if (x < 8) return 2;
        ...
        if (x < 2147483648) return 30;
}
```

Arrays

- Reduce array dimensions where possible

```
for (row = 0; row < numRows; row++) {
        for (column = 0; column < numColumns; column++) {
                matrix[row][column] = 0;
        }
}
```

  - Is faster as a 1D array

```
for (entry = 0; entry < numRows*numColumns; entry++) {
        matrix[entry] = 0;
}
```

- Minimize array references

```
for (i = 0; i < size; i++)
        for (j = 0; j < n; j++)
        rate[j] *= discount[i];

for (...) {
        temp = discount[i];;
        for (...)
                rate[j] *= temp;
}
```

- Use supplementary indices:
  - Length index or arrays
    - Add a string-length field to C strings
    - Faster than using `strlen()` which loops until `null` is found
  - Parallel index structure
    - Often easier to sort an array of references to a data array, then the data array itself
    - Avoids swapping data that's expensive to move (ie. is large or on disk)

Expressions

- Use caching: Save commonly used values, instead of recomputing or rereading them

```
private double cachedH = 0, cachedA = 0, cachedB = 0;
public double Hypotenuse(double A, double B) {
        if ((A == cachedA) && (B == cachedB)) { return cachedH; }
        cachedH = Math.sqrt((A*A) + (B*B));
        cachedA = A;
        cachedB = B;
        return cachedH;
}
```

- Expressions: Exploit algebraic identities
  - Replace expensive expressions with cheaper ones
  - `not a and not b = not (a or b)`
  - `if (sqrt(x) < sqrt(y)) = if (x < y)`
- Strength reduction

Original	Replacement
Multiplication	Repeated Addition
Exponentiation	Repeated Multiplication
Trig Routines	Trig Identities
Long Ints	Ints
Floats	Fixed Point Numbers/Ints
Doubles	Floats
Mult/Div by Power of 2	Left/Right Shift

- Initialize at compile time, use constants where possible

```
unsigned int Log2(unsigned int x) {
        return (unsigned int)(log(x) / log(2));
}

const double LOG2 = 0.69314718;
unsigned int Log2(unsigned int x) {
        return (unsigned int)(log(x) / LOG2);
}
```

- Use the proper data type for constants: Avoid runtime type conversion

```
double x;
...
x = 5;

// Better as
x = 5.0;
```

- Eliminate common subexpressions: Assign to a variable, use it instead of re-computing

```
p = (1.0 - (r / 12.0)) / (r / 12.0);

// Better as
y = r / 12.0;
p = (1.0 - y) / y;
```

- Precompute results: Often better to look up values than to recompute them
  - Values could be stored in constants, arrays, or files