Code Tuning
Deliverables

Logic Constructs

Various techniques to optimize code

Shortcuts in programming
Code Tuning-Introduction

Used to write better code

Needs better understanding of the programming language and its compiler

Equivalent of code optimization at higher language level
Important Constructs of Logic formulation

- If-Then-Else
- Switch Case
- While-Do
- Repeat-Until
- For Loop
- Exit Loop
If Then Else

- True: execute "true" action
- False: execute "false" action
- Actions to execute after IF-THEN-ELSE
Switch-Case
While-Do
Repeat Until
For Loop
Exit
Proving Correctness of Loops

Loop invariant: A statement that is always true about the loop.

To prove correctness proposition about a loop:

1. Identify loop invariant P.
2. Prove that initialization establishes truth of P.
3. Prove that execution of the loop body preserves the truth of P.
4. Prove that P and falsehood of loop condition, imply post-condition.
5. Prove that loop terminates.

Steps 1-4 ensure partial correctness of loop. 5 ensures total correctness.
## Comparative Analysis of various operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer add</td>
<td>X+Y</td>
<td>1 ns</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>X*Y</td>
<td>1 ns</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>X/Y</td>
<td>2 ns</td>
</tr>
<tr>
<td>Floating point add</td>
<td>X+Y</td>
<td>1.5 ns</td>
</tr>
<tr>
<td>Floating point Multiply</td>
<td>X*Y</td>
<td>1.5 ns</td>
</tr>
<tr>
<td>Floating point Divide</td>
<td>X/Y</td>
<td>6 ns</td>
</tr>
<tr>
<td>cos</td>
<td>Math. Cos(theta)</td>
<td>100 ns</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Increasing cost of operations

Variable Declaration → Assignment statement → Integer Compare → Array Element Access

String concatenation ← String length calculation ← 1D array allocation
Time calculation

Total Time to run our program will be
\[ c_1A + c_2 B + c_3 C + c_4 D + c_5 E + c_6 F \ldots \]
Where \( c_i \) is the cost of a particular operation which will vary from machine to machine and \( A, B \ldots \) is the number of times that operation appears in program under analysis.
## Space Requirement for common operations

<table>
<thead>
<tr>
<th>Data type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>1</td>
</tr>
<tr>
<td>Byte</td>
<td>1</td>
</tr>
<tr>
<td>Char</td>
<td>2</td>
</tr>
<tr>
<td>Int</td>
<td>4</td>
</tr>
<tr>
<td>Float</td>
<td>4</td>
</tr>
<tr>
<td>Long</td>
<td>8</td>
</tr>
<tr>
<td>Double</td>
<td>8</td>
</tr>
<tr>
<td>1D array Overhead</td>
<td>16 Bytes</td>
</tr>
<tr>
<td>Object overhead</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Pointer or reference overhead</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Loop Fusion and Loop Fission

After Loop fission

```c
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
}
for (i = 0; i < 100; i++)
{
    b[i] = 2;
}
```

After Loop fusion

```c
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i] = 2;
}
```
Loop Unrolling/Unwinding

Normal Code

```c
for (x = 0; x < 100; x++)
{
    print(x);
}
```

Code after unrolling

```c
for (x = 0; x < 100; x += 2)
{
    print(x);
    print(x+1);
}
```
Loop invariant code motion

Normal Code

```c
for (i = 0; i < n; ++i) {
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

Improved code

```c
x = y + z;
t1 = x * x;
for (i = 0; i < n; ++i) {
    a[i] = 6 * i + t1;
}
**Strength Reduction**

**Normal Code**

c = 8;
for (i = 0; i < N; i++)
{
    y[i] = c * i;
}

**Improved Code**

c = 8;
k = 0;
for (i = 0; i < N; i++)
{
    y[i] = k;
    k = k + c;
}
Minimize work inside loop

Normal code

```c
for (i = 1; i < n/2; i++)
{
...
}
```

Need to compute \( n/2 \) times in all iterations

Improved code

```c
m = n/2;
for (i = 1; i < m; i++)
{
...
}
```
Loop Peeling

Normal Code

```c
int p = 10;
for (int i=0; i<10; ++i)
{
    y[i] = x[i] + x[p];
p = i;
}
```

Improved Code

```c
y[0] = x[0] + x[10];
for (int i=1; i<10; ++i)
{
    y[i] = x[i] + x[i-1];
}
```
Normal Code

```c
int i, w, x[1000], y[1000];
for (i = 0; i < 1000; i++)
{
    x[i] = x[i] + y[i];
    if (w)=1
        y[i] = 0;
}
```

Unswitching Code

```c
int i, w, x[1000], y[1000];
if (w)
{
    for (i = 0; i < 1000; i++)
    {
        x[i] = x[i] + y[i];
        y[i] = 0;
    }
}
else
{
    for (i = 0; i < 1000; i++)
    {
        x[i] = x[i] + y[i];
    }
}
```
Use of sentinel Value

To find the $x$ in array $a$

```java
while(i<n) and (x<>a[i])
{
    i=i+1;
}
if (i<n)
    print number at position $i$
else
    print number not present
```

Improved Code

```java
a[n+1]=x;
while(x<>a[i])
{
    i=i+1;
}
if i==(n+1)
    print number not present
else
    print position of number $i$
```
Order Condition Testing

Normal Code
read(RegnNo)
case Grade(student)
1: {……}
2: {……}
3: {……}
4: {……}
endcase

Improved Code
read(RegnNo)
case Grade(student)
2: {……}
3: {……}
1: {……}
4: {……}
endcase

Applicable for if-else also
## Common expression elimination

### Normal Code

\[
a = b \times c + g;
d = b \times c \times d;
\]

### Improved Code

\[
tmp = b \times c;
a = tmp + g;
d = tmp \times d;
\]
Use constants of correct type

float x;
x=5;
convert 5 to 5.0 and then stores into x

int i;
i=3.14;
convert 3.14 to 3 and store into i
Precompute Results

Normal code

```c
for(i=0;i<100;i++)
{
    Y=log(i)/log(2)
    B=log(ni^a)/log(2)
}
```

Improved Code

```c
twolog=log(2)
for(i=0;i<100;i++)
{
    Y=log(i)/ twolog
    B=log(ni^a)/ twolog
}
```
Dead Code elimination

int xyz()
{
    int a = 24;
    int b = 25; /*dead variable
    int c;
    c = a - 2;
    return c;
    b = 24; /* Unreachable
    return 0;
}

Improved Code

int xyz()
{
    int a = 24;
    int c;
    c = a - 2;
    return c;
}
Lazy computations

Normal code
n = x*x + 2*y + z
if q > 10 then
{
    ....
}
else q > n then
{
    ....
}
else
{
    ...

Improved code
if q > 10 then
{
    ....
} else
{
    n = x*x + 2*y + z
    if q > n then
    {
        ....
    } else
    {
        ....
    }
}

6/5/2012 7:12 PM
Exploit algebraic identities

- Algebraic identities can be used to replace costlier operations by cheaper ones.
- To find whether $\sqrt{x} < \sqrt{y}$, we can use the algebraic identity which says $x < y$ only when $x < y$. So it is enough to check if $x < y$ in this case.
- $\neg (A \text{ or } B)$ is cheaper than $\neg A$ and $\neg B$. 
Short circuiting and reordering

A particular condition may have the probability of being true or false

```java
if (a>b) && (c>d) && (e>f)
{
    ....
}
```
Minimizing array references

Normal code

```c
for( a=0; a<5; a++ )
{
    for (b=0; b<10; b++)
    {
        Total[b]=total[b]*sum[a]
    }
}
```

Improved code

```c
for( a=0; a<5; a++ )
{
    Sum_t=sum[a]
    for (b=0; b<10; b++)
    {
        Total[b]=total[b]*sum_t;
    }
}
```
Locality of Reference

Row major, cache-friendly, use this method to access all items in array sequentially.

for (i=0; i<numRow; i++) // RowFirst = Effective
for (j=0; j<numCol; j++)
array[i][j] = 0;

Column major, NOT cache-friendly, very poor performance. because computer access array data in row major.

for (j=0; j<numCol; j++) // ColFirst = Ineffective
for (i=0; i<numRow; i++)
array[i][j] = 0;
Questions and Suggestions and Comments
Question 1

What will be the output of the following program

```c
main()
{
 int count, i, x;
 for(count=1,x=0,i=0; count<=4;count++,i++)
 {
  x=i%2;
  if(x==0)
   continue;
  else
   {
   printf("%d\n", x);
   continue;
   }
}
printf("%d\n", count);
}
```
Question 2

For calculating the sum of 100 numbers stored in the array we have following code
for (int x = 0; x < 100; x++)
{
    S=S+a[x];
}

Above code can be replaced by one of the following, which will involve less operations
A)
x=0;
a=value=A[x];
for (int x = 0; x < 100; x++)
{
    S=S+a[x];
}
B) for (int x = 0; x < 100; x=x+2)
{
    S=S+a[x]+(x+1);
}
C) for (int x = 0; x < 100; x=x+3)
{
    S=S+a[x]+a[x+1]+a[x+2];
}
D) for (int x = 0; x < 100; x=x+2)
{
    S=S+a[x]+a[x+1];
}
Question 3

Which One is a better data structure in context of locality of reference or cache performance.
A) Array
B) Link List
C) Stack
D) Queue