Lecture 2
Bentley Rules for Optimizing Work
Charles E. Leiserson
September 11, 2012
Definition.
The work of a program (on a given input) is the sum total of all the operations executed by the program.
Optimizing Work

- Algorithm design can produce dramatic reductions in the amount of work it takes to solve a problem, as when a $\Theta(n \lg n)$-time sort replaces a $\Theta(n^2)$-time sort.
- Reducing the work of a program does not automatically reduce its running time, however, due to the complex nature of computer hardware:
  - instruction-level parallelism (ILP),
  - caching,
  - vectorization,
  - speculation and branch prediction,
  - etc.
- Nevertheless, reducing the work serves as a good heuristic for reducing overall running time.
“BENTLEY” OPTIMIZATION RULES
Jon Louis Bentley

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
New Bentley Rules

- Most of Bentley’s original rules dealt with work, but some dealt with the vagaries of computer architecture three decades ago.
- We have created a new set of Bentley rules dealing only with work.
- We shall discuss architecture–dependent optimizations in subsequent lectures.
New “Bentley” Rules

Data structures
- Packing and encoding
- Augmentation
- Precomputation
- Compile-time initialization
- Caching
- Sparsity

Loops
- Hoisting
- Sentinels
- Loop unrolling
- Loop fusion
- Eliminating wasted iterations

Logic
- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Short-circuiting
- Ordering tests
- Combining tests

Functions
- Inlining
- Tail-recursion elimination
- Coarsening recursion
The idea of **packing** is to store more than one data value in a machine word. The related idea of **encoding** is to convert data values into a representation requiring fewer bits.

**Example**: Encoding dates

- The string “February 14, 2008” can be stored in 19 bytes (null terminating byte included), which means that 3 double (64-bit) words must moved whenever a date is manipulated using this representation.
- Assuming that we only store years between 1 C.E. and 4096 C.E., there are about $365.25 \times 4096 \approx 1.5 \text{ M}$ dates, which can be encoded in $\lceil \lg(1.5 \times 10^6) \rceil = 21$ bits, which fits in a single (32-bit) word.
- But querying the month of a date takes more work.
Example: Packing dates

- Instead, let us pack the three fields into a word:

  ```c
  typedef struct {
      unsigned int year: 12;
      unsigned int month: 4;
      unsigned int day: 5;
  } date_t;
  ```

- This packed representation still only takes 21 bits, but the individual fields can be extracted much more quickly than if we had encoded the 1.5 M dates as sequential integers.

Sometimes **unpacking** and **decoding** are the optimization, depending on whether more work is involved moving the data or operating on it.
Augmentation

The idea of data-structure augmentation is to add information to a data structure to make common operations do less work.

**Example:**Appending singly linked lists

- Appending one list to another requires walking the length of the first list to set its null pointer to the start of the second.

- **Augmenting** the list with a tail pointer allows appending to operate in constant time.
The idea of precomputation is to perform calculations in advance so as to avoid doing them at “mission-critical” times.

**Example:** Binomial coefficients

\[
\binom{a}{b} = \frac{a!}{b!(a-b)!}
\]

Expensive to compute (lots of multiplications), and watch out for integer overflow for even modest values of \(a\) and \(b\).

**Idea:** Precompute the table of coefficients when initializing, and do table look-up at runtime.
Precomputation (2)

Pascal’s triangle

```c
#define CHOOSE_SIZE 100
unsigned int choose[CHOOSE_SIZE][CHOOSE_SIZE];

void init_choose() {
    for (int n=0; n<CHOOSE_SIZE; ++n) {
        choose[n][0] = 1;
        choose[n][n] = 1;
    }
    for (int n=1; n<CHOOSE_SIZE; ++n) {
        choose[0][n] = 0;
        for (int k=1; k<n; ++k) {
            choose[n][k] = choose[n-1][k-1] + choose[n-1][k];
            choose[k][n] = 0;
        }
    }
}
```
Compile–Time Initialization

The idea of compile–time initialization is to store the values of constants during compilation, saving work at execution time.

Example

```c
unsigned int choose[10][10] = {
    { 1,  0,  0,  0,  0,  0,  0,  0,  0,  0, },
    { 1,  1,  0,  0,  0,  0,  0,  0,  0,  0, },
    { 1,  2,  1,  0,  0,  0,  0,  0,  0,  0, },
    { 1,  3,  3,  1,  0,  0,  0,  0,  0,  0, },
    { 1,  4,  6,  4,  1,  0,  0,  0,  0,  0, },
    { 1,  5, 10, 10, 5,  1,  0,  0,  0,  0, },
    { 1,  6, 15, 20, 15, 6,  1,  0,  0,  0, },
    { 1,  7, 21, 35, 35, 21, 7,  1,  0,  0, },
    { 1,  8, 28, 56, 70, 56, 28, 8,  1,  0, },
    { 1,  9, 36, 84, 126, 126, 84, 36, 9,  1, },
};
```
Idea: Create large static tables by metaprogramming.

```c
int main(int argc, const char *argv[]) {
    init_choose();
    printf("unsigned int choose[10][10] = {\n");
    for (int a = 0; a < 10; ++a) {
        printf("  {\n          for (int b = 0; b < 10; ++b) {
            printf("%3d, ", choose[a][b]);
        }
        printf("\n");
    }
    printf("};\n");
}
```
The idea of caching is to store results that have been accessed recently so that the program need not compute them again.

```c
inline double hypotenuse(double A, double B) {
    return sqrt(A * A + B * B);
}
```

```c
double cached_A = 0.0;
double cached_B = 0.0;
double cached_h = 0.0;

inline double hypotenuse(double A, double B) {
    if (A == cached_A && B == cached_B) {
        return cached_h;
    }
    cached_A = A;
    cached_B = B;
    cached_h = sqrt(A * A + B * B);
    return cached_h;
}
```

About 30% faster if cache is hit 2/3 of the time.
Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes. “The fastest way to compute is not to compute at all.”

Example: Sparse matrix multiplication

\[
\begin{pmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
5 & 0 & 0 & 0 & 8 & 0 \\
0 & 0 & 0 & 9 & 7 & 0
\end{pmatrix}
\begin{pmatrix}
1 \\
4 \\
2 \\
8 \\
5 \\
7
\end{pmatrix}
\]

Dense matrix–vector multiplication performs \(n^2 = 36\) scalar multiplies, but only 14 entries are nonzero.
Sparsity (2)

Compressed Sparse Rows (CSR)

rows: 0 2 6 8 10 11 14

cols: 0 4 1 2 4 5 3 5 0 3 0 4 3 4
vals: 3 1 4 1 5 9 2 6 5 3 5 8 9 7

\[
\begin{pmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 8 & 9 & 7 \\
\end{pmatrix}
\]

\(n = 6\)
\(\text{nnz} = 14\)
Sparsity (3)

CSR matrix–vector multiplication

```c
typedef struct {
    int n, nnz;
    int *rows; // length n
    int *cols; // length nnz
    double *vals; // length nnz
} sparse_matrix_t;

void spmv(sparse_matrix_t *A, double *x, double *y) {
    for (int i = 0; i < A->n; i++) {
        y[i] = 0;
        for (int k = A->rows[i]; k < A->rows[i+1]; k++) {
            int j = A->cols[k];
            y[i] += A->vals[k] * x[j];
        }
    }
}
```

Number of scalar multiplications = nnz, which is potentially much less than $n^2$. 
LOGIC

SPEED LIMIT

PER ORDER OF 6.172

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
The idea of **constant folding and propagation** is to evaluate constant expressions and substitute the result into further expressions, all during compilation.

```c
#include <math.h>

void orrery() {
    const double radius = 6371000.0;
    const double diameter = 2 * radius;
    const double circumference = M_PI * diameter;
    const double cross_area = M_PI * radius * radius;
    const double surface_area = circumference * diameter;
    const double volume = 4 * M_PI * radius * radius * radius / 3;
    // ...
}
```

With a sufficiently high optimization level, all the expressions are evaluated at compile-time.
The idea of **common-subexpression elimination** is to avoid computing the same expression multiple times by evaluating the expression once and storing the result for later use.

```plaintext
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

The third line cannot be replaced by `c = a`, because the value of `b` changes in the second line.
Algebraic Identities

The idea of **exploiting algebraic identities** is to replace expensive logical expressions with algebraic equivalents that require less work.

```c
#include <stdbool.h>
#include <math.h>

typedef struct {
    double x;  // x-coordinate
    double y;  // y-coordinate
    double z;  // z-coordinate
    double r;  // radius of ball
} ball_t;

double square(double x) {
    return x * x;
}

bool collides(ball_t *b1, ball_t *b2) {
    double d = sqrt(square(b1->x - b2->x)
        + square(b1->y - b2->y)
        + square(b1->z - b2->z));
    return d <= b1->r + b2->r;
}
```
The idea of exploiting algebraic identities is to replace expensive logical expressions with algebraic equivalents that require less work.

\[ \sqrt{u} \leq v \] exactly when
\[ u \leq v^2. \]

```c
#include <stdbool.h>
#include <math.h>

typedef struct {
    double x;  // x-coordinate
    double y;  // y-coordinate
    double z;  // z-coordinate
    double r;  // radius of ball
} ball_t;

double square(double x) {
    return x * x;
}

bool collides(ball_t *b1, ball_t *b2) {
    double dsquared = square(b1->x - b2->x)
        + square(b1->y - b2->y)
        + square(b1->z - b2->z);
    return dsquared <= square(b1->r + b2->r);
}
```
Short-Circuiting

When performing a series of tests, the idea of short-circuiting is to stop evaluating as soon as you know the answer.

```c
#include <stdbool.h>

bool sum_exceeds(int *A, int n, int limit) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum > limit;
}
```

Note that `&&` and `||` are short-circuiting logical operators.
Ordering Tests

Consider code that executes a sequence of logical tests. The idea of ordering tests is to perform those that are more often “successful” — a particular alternative is selected by the test — before tests that are rarely successful. Similarly, inexpensive tests should precede expensive ones.

```c
#include <stdbool.h>
bool is_whitespace(char c) {
    if (c == '\0' || c == ' ' || c == '
') {
        return true;
    }
    return false;
}
```

The idea of combining tests is to replace a sequence of tests with one test or switch.

### Full adder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>carry</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
void full_add (int a,
  int b,
  int c,
  int *sum,
  int *carry) {
  if (a == 0) {
    if (b == 0) {
      if (c == 0) {
        *sum = 0;
        *carry = 0;
      } else {
        *sum = 1;
        *carry = 0;
      }
    } else {
      *sum = 1;
      *carry = 1;
    }
  } else {
    if (b == 0) {
      if (c == 0) {
        *sum = 0;
        *carry = 0;
      } else {
        *sum = 0;
        *carry = 1;
      }
    } else {
      *sum = 1;
      *carry = 1;
    }
  }
}
```
The idea of **combining tests** is to replace a sequence of tests with one test or switch.

### Full adder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>carry</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

For this example, table look-up is even better!

```c
void full_add (int a, int b, int c, int *sum, int *carry) {
    int test = ((a == 1) << 2) | ((b == 1) << 1) | (c == 1);
    switch(test) {
        case 0:
            *sum = 0;
            *carry = 0;
            break;
        case 1:
            *sum = 1;
            *carry = 0;
            break;
        case 2:
            *sum = 1;
            *carry = 0;
            break;
        case 3:
            *sum = 0;
            *carry = 1;
            break;
        case 4:
            *sum = 1;
            *carry = 0;
            break;
        case 5:
            *sum = 0;
            *carry = 1;
            break;
        case 6:
            *sum = 0;
            *carry = 1;
            break;
        case 7:
            *sum = 1;
            *carry = 1;
            break;
    }
}
```
SPEED LIMIT
PER ORDER OF 6.172

LOOPS
Hoisting

The goal of hoisting — also called loop-invariant code motion — is to avoid recomputing loop-invariant code each time through the body of a loop.

```c
#include <math.h>

void scale(double *X, double *Y, int N) {
    for (int i = 0; i < N; i++) {
        Y[i] = X[i] * exp(sqrt(M_PI/2));
    }
}
```

```c
#include <math.h>

void scale(double *X, double *Y, int N) {
    double factor = exp(sqrt(M_PI/2));
    for (int i = 0; i < N; i++) {
        Y[i] = X[i] * factor;
    }
}
```
Sentinels are special dummy values placed in a data structure to simplify the logic of boundary conditions, and in particular, the handling of loop-exit tests.

```c
#include <stdint.h>
#include <stdbool.h>

bool overflow(uint64_t *A, size_t n) {
    uint64_t sum = 0;
    for (size_t i = 0; i < n; ++i)
        sum += A[i];
    if (sum < A[i]) return true;
    return false;
}
```

```c
#include <stdint.h>
#include <stdbool.h>

// Assumes that A[n] and A[n+1] exist and can be clobbered
bool overflow(uint64_t *A, size_t n) {
    A[n] = UINT64_MAX;
    A[n+1] = 1; // or any positive number
    size_t i = 0;
    uint64_t sum = A[0];
    while (sum >= A[i]) {
        sum += A[++i];
    }
    if (i < n) return true;
    return false;
}
```
Loop Unrolling

Loop unrolling attempts to save work by combining several consecutive iterations of a loop into a single iteration, thereby reducing the total number of iterations of the loop and, consequently, the number of times that the instructions that control the loop must be executed.

- **Full** loop unrolling: All iterations are unrolled.
- **Partial** loop unrolling: Several, but not all, of the iterations are unrolled.
Full Loop Unrolling

```c
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += A[i];
}
```

```c
int sum = 0;
sum += A[0];
sum += A[1];
sum += A[2];
sum += A[3];
sum += A[4];
sum += A[5];
sum += A[6];
sum += A[7];
sum += A[8];
sum += A[9];
```
Partial Loop Unrolling

```c
int sum = 0;
for (int i = 0; i < n; ++i) {
  sum += A[i];
}
```

```c
int sum = 0;
int j;
for (j = 0; j < n - 3; j += 4) {
  sum += A[j];
  sum += A[j+1];
  sum += A[j+2];
  sum += A[j+3];
}
for (int i = j; i < n; ++i) {
  sum += A[i];
}
```
The idea of loop fusion — also called jamming — is to combine multiple loops over the same index range into a single loop body, thereby saving the overhead of loop control.

```c
for (int i = 0; i < n; ++i) {
}
for (int i = 0; i < n; ++i) {
}
for (int i = 0; i < n; ++i) {
}
```
Eliminating Wasted Iterations

The idea of **eliminating wasted iterations** is to modify loop bounds to avoid executing loop iterations over essentially empty loop bodies.

```c
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (i < j) {
            int temp = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = temp;
        }
    }
}
```
FUNCTIONS

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
The idea of inlining is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself.

```c
double square(double x) {
    return x * x;
}

double sum_of_squares(double *A, int n) {
    double sum = 0.0;
    for (int i = 0; i < n; ++i) {
        sum += square(A[i]);
    }
    return sum;
}
```
The idea of inlining is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself.

```
double square(double x) {
    return x * x;
}

double sum_of_squares(double *A, int n) {
    double sum = 0.0;
    for (int i = 0; i < n; ++i) {
        sum += square(A[i]);
    }
    return sum;
}
```

Inlined functions can be just as efficient as macros, and they are better structured.

```
inline double square(double x) {
    return x * x;
}

double sum_of_squares(double *A, int n) {
    double sum = 0.0;
    for (int i = 0; i < n; ++i) {
        sum += square(A[i]);
    }
    return sum;
}
```
The idea of **tail-recursion elimination** is to replace a recursive call that occurs as the last step of a function with a branch, saving function-call overhead.

```c
void quicksort(int *A, int n) {
    if (n > 1) {
        int r = partition(A, n);
        quicksort(A, r);
        quicksort(A + r + 1, n - r - 1);
    }
}
```

```c
void quicksort(int *A, int n) {
    while (n > 1) {
        int r = partition(A, n);
        quicksort(A, r);
        A += r + 1;
        n -= r + 1;
    }
}
```
Coarsening Recursion

The idea of **coarsening recursion** is to increase the size of the base case and handle it with more efficient code that avoids function-call overhead.

```c
void quicksort(int *A, int n) {
    while (n > 1) {
        int r = partition(A, n);
        quicksort(A, r);
        A += r + 1;
        n -= r + 1;
    }
}
```

```c
#define THRESHOLD 10
void quicksort(int *A, int n) {
    while (n > THRESHOLD) {
        int r = partition(A, n);
        quicksort(A, r);
        A += r + 1;
        n -= r + 1;
    }
    // insertion sort for small arrays
    for (int j = 1; j < n; ++j) {
        int key = A[j];
        int i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i+1] = A[i];
            --i;
        }
        A[i+1] = key;
    }
}
```
SUMMARY
Bentley Rules

Data structures
- Packing and encoding
- Augmentation
- Precomputation
- Compile-time initialization
- Caching
- Sparsity

Loops
- Hoisting
- Sentinels
- Loop unrolling
- Loop fusion
- Eliminating wasted iterations

Logic
- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Short-circuiting
- Ordering tests
- Combining tests

Functions
- Inlining
- Tail-recursion elimination
- Coarsening recursion
Conclusions

- Avoid premature optimization. First get correct working code. Then optimize.
- Reducing the work of a program does not necessarily decrease its running time, but it is a good heuristic.
- The compiler automates many low-level optimizations.
- To tell whether the compiler is actually automating a particular optimization, look at the assembly code (next lecture).

If you find interesting examples of work optimization, please let me know!