Software Performance Engineering

LECTURE 3

Bentley Rules
for optimizing **Work**

Xuhao Chen

Wednesday, September 3, 2025

A Situation

Situation

You and your partner discover a clever algorithm, compiler switch, etc. that speeds up your code.

Q. Which of the following would you do?

a. Keep it secret so that you can beat the other teams.
b. Publish the idea on Piazza.

Course Policy

1. You are not competing with other teams. The cutoffs for grades are determined independently of how teams perform.
2. You receive class-contribution points for sharing ideas and code snippets on Piazza
3. You may not copy code, but you can take inspiration from each other.

Work

Definition.
The work of a program (on a given input) is the sum total of all the operations executed by the program.

The number of executed instructions

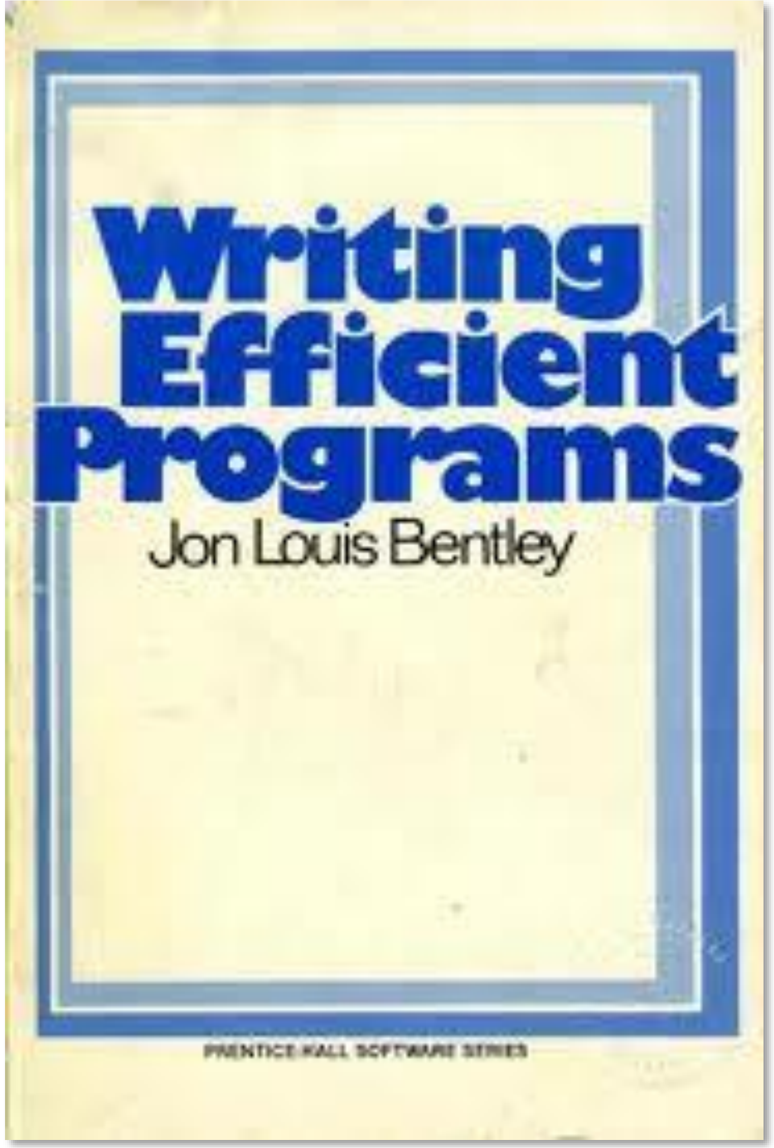
Avoid unnecessary work

Reducing Work

- Less work $\approx$ faster code
- Reducing the work of a program does not automatically reduce its running time, due to the complex nature of computer hardware:
  - instruction-level parallelism (ILP),
  - caching,
  - vectorization,
  - speculation and branch prediction, etc
- But reducing work is a good heuristic for reducing overall exec. time
- Algorithm design can produce dramatic reductions in the work
  - e.g. when a $\Theta(n \lg n)$-time sort replaces a $\Theta(n^2)$-time sort

Bentley Rules for Optimizing Work

Jon Louis Bentley



1982

New Bentley Rules

Data structures

Loops

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

Loops

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

Logic

Functions

- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Creating a fast path
- Short-circuiting
- Ordering tests
- Combining tests

Functions

- Inlining
- Tail-recursion elimination
- Coarsening recursion

DATA STRUCTURES

Packing and Encoding

Packing is to store more than one data value in a machine word.
Encoding is to convert data into a representation that requires fewer bits

Example: Encoding dates

- The string "`September 3, 2020`" can be stored in 17 bytes — more than two 64-bit words — which must move whenever the date is manipulated.
- Assuming that we only store dates between 4096 B.C.E. and 4096 C.E., there are about $365.25 \times 8192 \approx 3\,\mathrm{M}$ dates, which can be encoded in $\lceil \lg(3\times 10^6) \rceil = 22$ bits, easily fitting in a 32-bit word.
- **Problem:** How can we represent dates compactly so that determining the year, month, and day is fast?

Packing and Encoding (2)

Example: Packing dates

- Let us pack the three fields into a word:

```c
typedef struct {
  int year: 13;
  int month: 4;
  int day: 5;
} date_t;
```

- Still only takes 22 bits

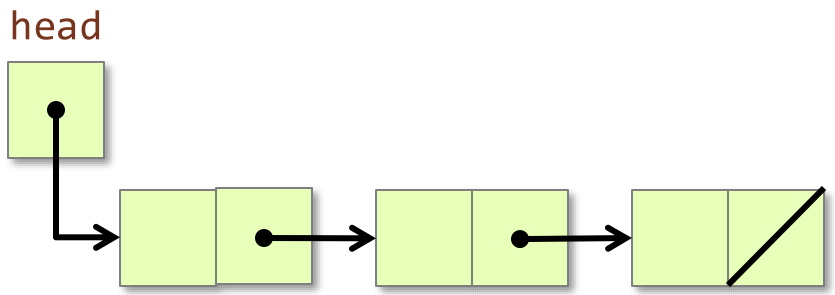
- But individual fields can be extracted much more quickly

Augmentation

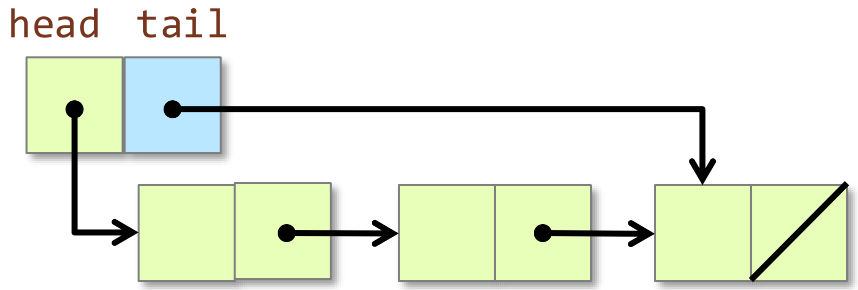
The idea is to **add information** to a data structure to make common operations do less work.

Example: Appending one singly linked list to another

- Walk through the first list, and set its null pointer to the start of the second



- Augmenting the list with a **tail** pointer allows appending to operate in constant time.



Caching

The idea is to **store results** that have been accessed recently so that the program need not compute them again.

Before

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

After

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

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.

Precomputation

The idea is to **perform calculations in advance** so as to avoid doing them at "mission-critical" times.

Example: Binomial coefficients

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- Expensive (lots of multiplications)
- Integer **overflow** for even modest values of n and k

Idea: Precompute the table of coefficients when initializing, and perform table look-up at runtime.

Step 1: Pascal's Triangle

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

k

n

1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0
1	3	3	1	0	0	0	0	0
1	4	6	4	1	0	0	0	0
1	5	10	10	5	1	0	0	0
1	6	15	20	15	6	1	0	0
1	7	21	35	35	21	7	1	0
1	8	28	56	70	56	28	8	1

```c
int choose(int n, int k) {
  if (n < k)  return 0;
  if (k == 0) return 1;
  return choose(n-1, k-1) + choose(n-1, k);
}
```

Step 2: Precomputing Pascal

```c
#define CHOOSE_SIZE 100
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;
    }
  }
}
```

Now, whenever we need a binomial coefficient (less than 100), we can simply index the choose array.

Compile-Time Initialization

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

Example

```
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, },
};
```

Compile-Time Initialization (2)

Idea: Create large static tables by metaprogramming.

```
#define N 100
int main(int argc, const char *argv[]) {
  init_choose();
  printf("#define N %3d\n”, N);
  printf("int choose[N][N] = {\n");
  for (int a = 0; a < N; ++a) {
    printf("  {");
    for (int b = 0; b < N; ++b) {
      printf("%3d, ", choose[a][b]);
    }
    printf("},\n");
  }
  printf("};\n");
}
```

Sparsity

The idea is to **avoid** storing and computing on **zeroes**.
"The fastest way to compute is not to compute at all."

Example: Matrix-vector multiplication

$$y = \begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 4 & 1 & 0 & 5 & 9 \\ 0 & 0 & 0 & 2 & 0 & 6 \\ 5 & 0 & 0 & 3 & 0 & 0 \\ 5 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 7 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 2 \\ 8 \\ 5 \\ 7 \end{pmatrix}$$

Dense matrix-vector multiplication performs $\mathbf{n^2 = 36}$ scalar multiplies, but only **14** entries are nonzero.

Sparsity

The idea is to **avoid** storing and computing on **zeroes**.
"The fastest way to compute is not to compute at all."

Example: Matrix-vector multiplication

$$y = \begin{pmatrix} 3 & & & & 1 & \\ & 4 & 1 & & 5 & 9 \\ & & & 2 & & 6 \\ 5 & & & 3 & & \\ 5 & & & & 8 & \\ & & & 9 & 7 & \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 2 \\ 8 \\ 5 \\ 7 \end{pmatrix}$$

Dense matrix-vector multiplication performs $n^2 = 36$ scalar multiplies, but only **14** entries are nonzero.

Sparsity (2)

Compressed Sparse Rows (CSR)

```
          0   1   2   3   4   5   6   7   8   9   10  11  12  13
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]
```

$$
\begin{array}{c c}
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 8 & 9 & 7
\end{bmatrix} \\
 & \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}
\end{array}
$$

$n = 6$

$nnz = 14$

Storage is $O(n+nnz)$ instead of n^2

Sparsity (3)

CSR matrix-vector multiplication

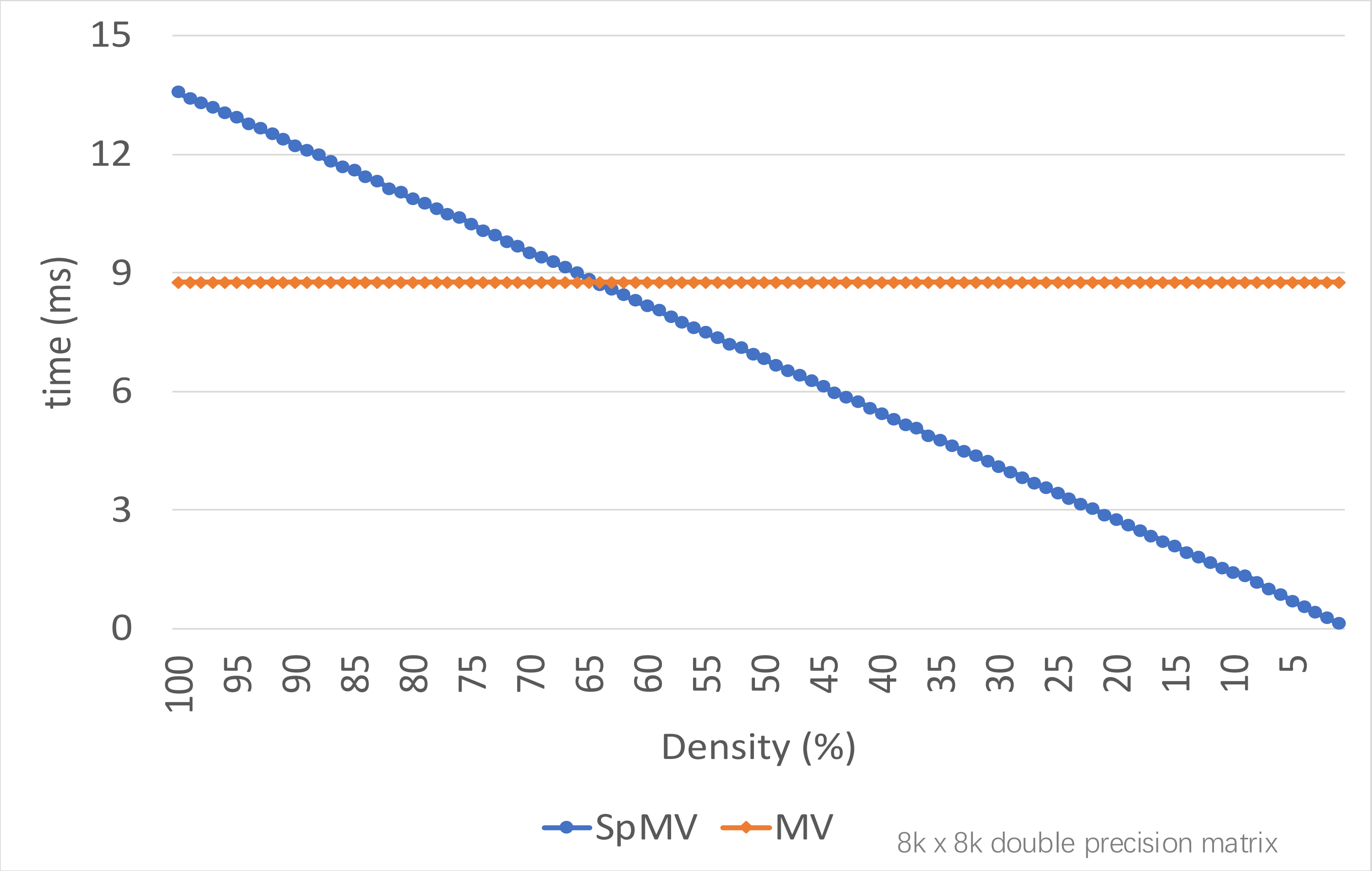
```
typedef struct {
  int n, nnz;
  int *rows;      // length n
  int *cols;      // length nnz
  double *vals;   // length nnz
} sparse_matrix_t;

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.

See the **TACO** research project if you are interested (https://tensor-compiler.org/)

Sparsity (3)

CSR matrix-vector multiplication

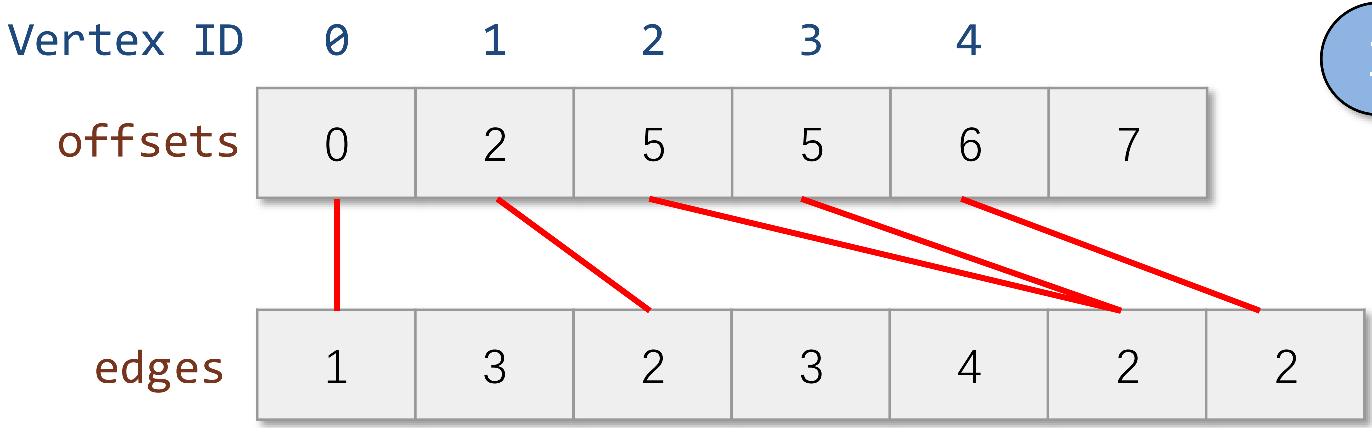


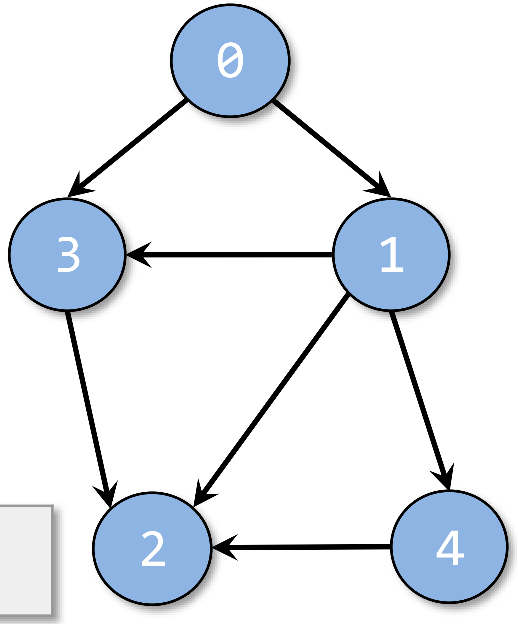
Number of scalar multiplications = nnz, which is potentially much less than n^2.

See the **TACO** research project if you are interested (https://tensor-compiler.org/)

Sparsity (4)

Storing a sparse graph





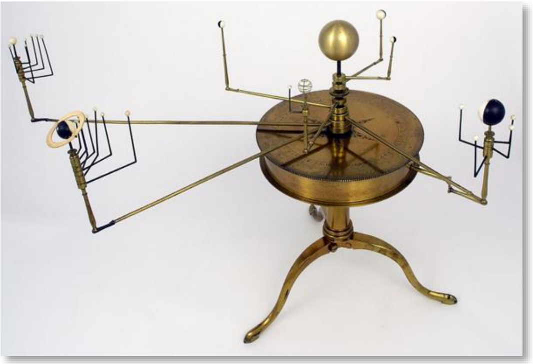
- Many graph algorithms run efficiently on this representation, e.g., breadth-first search, PageRank.

Logic

Constant Folding and Propagation

The idea of 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;
  // ...
}
```



mechanical orrery[1]

With a sufficiently high optimization level, all the expressions are evaluated at compile-time.

[1] https://en.wikipedia.org/wiki/Orrery#/media/File:Thinktank_Birmingham_-_object_1956S00682.00001(1).jpg

Common-Subexpression Elimination

The idea is to avoid computing the same expression multiple times by evaluating the expression once and saving the result for later use

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

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

Common-Subexpression Elimination

The idea is to avoid computing the same expression multiple times by **evaluating** the expression **once** and **saving** the result for later use

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

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

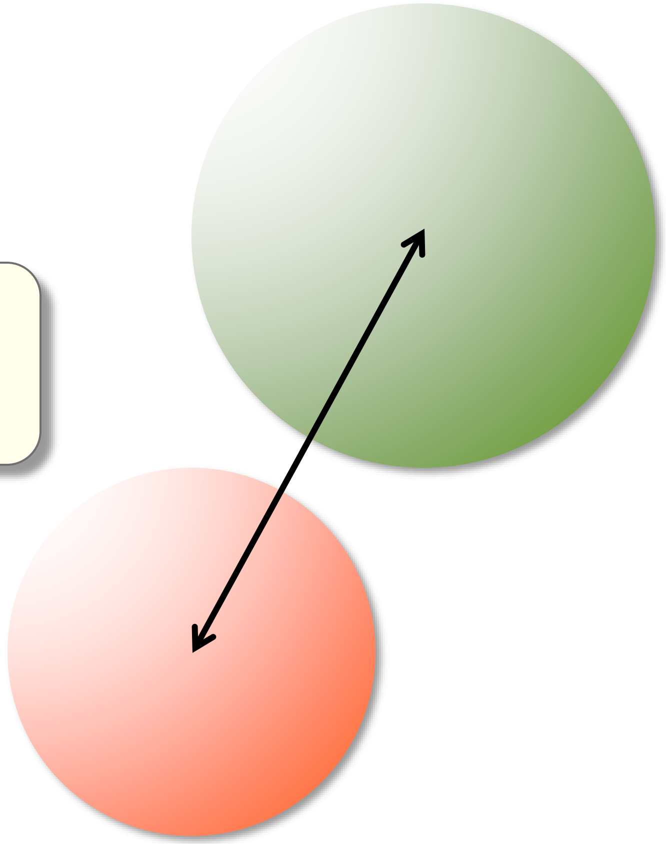
The third line cannot be replaced by `c = a`, because the value of `b` changes in the second line

Algebraic Identities

The idea is to **replace** expensive algebraic expressions with algebraic equivalents that require less work

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

typedef struct {
  double x, y, z;  // spatial coordinates
  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;
}
```

Expensive routine!



Algebraic Identities

The idea is to **replace** expensive algebraic expressions with algebraic equivalents that require less work

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

typedef struct {
  double x, y, z;  // spatial coordinates
  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;
}
```

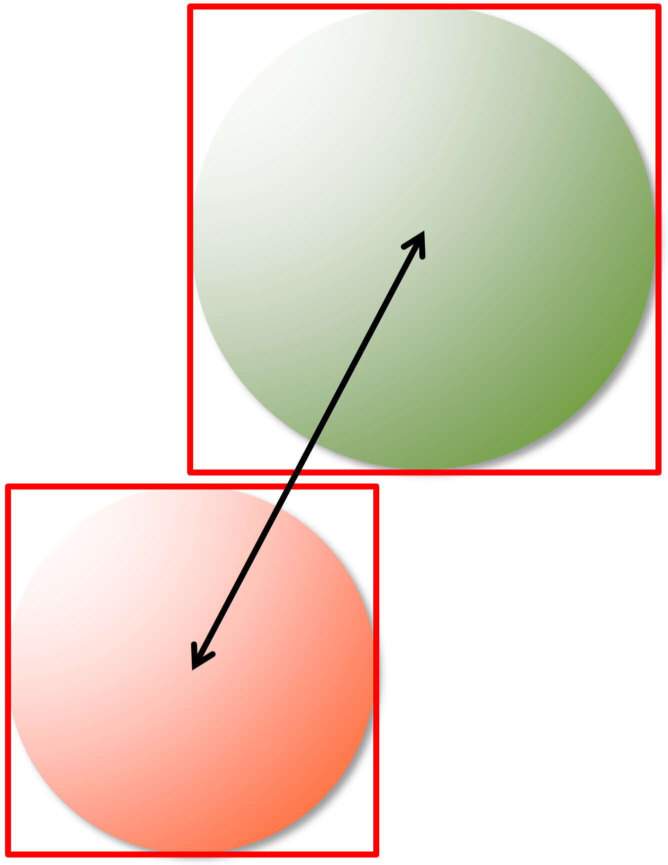
$\sqrt{u} \leq v$ exactly when $u \leq v^2$

```
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);
}
```

Caution: Be careful with floating point!

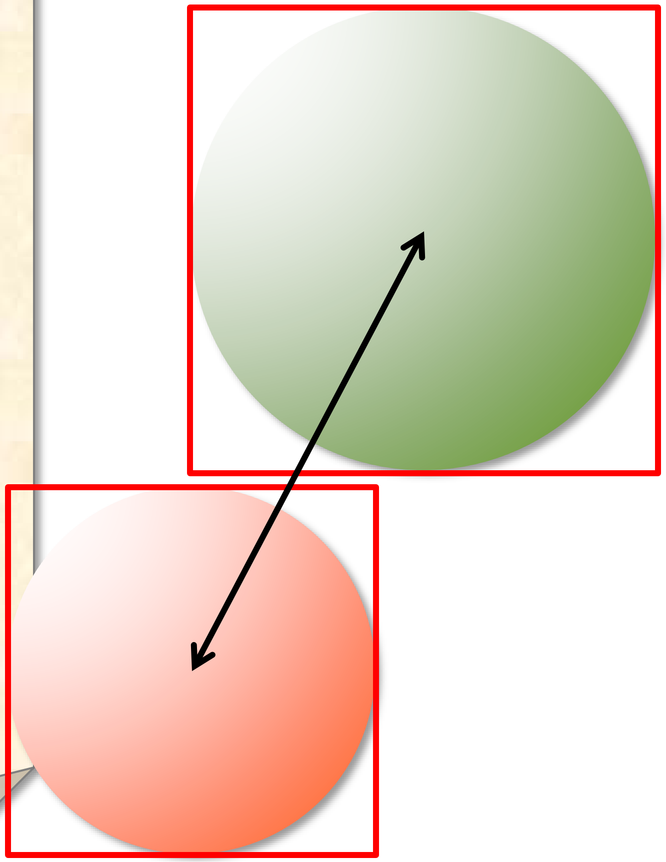
Creating a Fast Path

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

typedef struct {
  double x, y, z;   // spatial coordinates
  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);
}
```



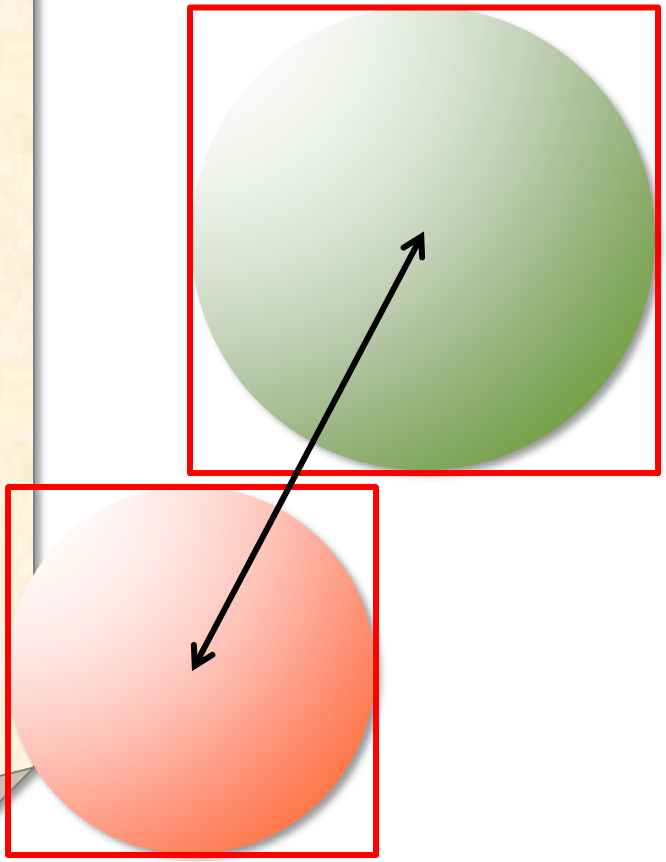
Creating a Fast Path

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

typedef struct {
  double x, y, z;  // spatial coordinates
  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);
}
```



Creating a Fast Path

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

typedef struct {
  double x, y, z;  // spatial coordinates
  double r;        // radius of ball
} ball_t;

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

bool collides(ball_t *b1, ball_t *b2) {
  if ((abs(b1->x – b2->x) > (b1->r + b2->r)) ||
      (abs(b1->y – b2->y) > (b1->r + b2->r)) ||
      (abs(b1->z – b2->z) > (b1->r + b2->r)))
    return false;
  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

The idea is to **stop** evaluating **as soon as** you know the answer, when performing a series of tests

```
#include <stdbool.h>
// All elements of A are nonnegative
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;
}
```

Short-Circuiting

The idea is to **stop** evaluating **as soon as** you know the answer, when performing a series of tests

Before

```c
#include <stdbool.h>
// All elements of A are nonnegative
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;
}
```

After

```c
#include <stdbool.h>
// All elements of A are nonnegative
bool sum_exceeds(int *A, int n, int limit) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += A[i];
    if (sum > limit) {
      return true;
    }
  }
  return false;
}
```

Ordering Tests

Consider code that executes a sequence of logical tests. The idea is to perform those that are more often "successful" **before** tests that are rarely successful.

Before

```c
#include <stdbool.h>
bool is_whitespace(char c) {
  return (c == '\r' || c == '\t' || c == ' ' || c == '\n');
}
```

After

```c
#include <stdbool.h>
bool is_whitespace(char c) {
  return (c == ' ' || c == '\n' || c == '\t' || c == '\r');
}
```

Note that && and || are short-circuiting logical operators, whereas & and | are not.

Combining Tests

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

Full adder

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

```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 {
      if (c == 0) {
        *sum = 1;
        *carry = 0;
      } else {
        *sum = 0;
        *carry = 1;
      }
    }
  } else {
    if (b == 0) {
      if (c == 0) {
        *sum = 1;
        *carry = 0;
      } else {
        *sum = 0;
        *carry = 1;
      }
    } else {
      if (c == 0) {
        *sum = 0;
        *carry = 1;
      } else {
        *sum = 1;
        *carry = 1;
      }
    }
  }
}
```

Combining Tests (2)

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

Full adder

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

In this case, the outputs can be computed mathematically.

```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;
  }
}
```

Loops

Why Loops?

Loops are often the focus of performance optimization. Why?

Loops account for a lot of work!

Consider this thought experiment:

- Suppose that a 2 GHz processor can execute 1 instruction per clock cycle.
- Suppose that a program contains 16 GB of instructions, but it is all simple straight-line code, i.e., no backwards branches.
- **Question:** How long does the code take to run?

Answer: at most 8 seconds!

What Happens When a Loop Runs?

Pseudocode for loop execution

```
int sum = 0;
int i = 0;
if (i >= N)
  goto loop_exit;
sum += A[i];
i++;
if (i >= N)
  goto loop_exit;
sum += A[i];
i++;
if (i >= N)
  goto loop_exit;
sum += A[i];
i++;
if (i >= N)
  goto loop_exit;
sum += A[i];
i++;
if (i >= N)
  goto loop_exit;
// ...
```

A simple loop

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

Loop control

Loop Unrolling

The idea is to save work by combining consecutive iterations of a loop into a single iteration

- Full loop unrolling: All iterations are unrolled.

- Partial loop unrolling: Several, but not all, of the iterations are unrolled.

Loop unrolling reduces 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

Before

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

After

```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

Before

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

After

```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];
}
```

Benefits of loop unrolling

- Fewer instructions devoted to loop control.
- Enables more compiler optimizations.

Caution: Unrolling too much can cause poor use of the instruction cache, because the code is bigger.

Hoisting

The idea is to avoid recomputing loop-invariant code each time through the body of a loop. a.k.a. loop-invariant code motion

Before

```
#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));
  }
}
```

After

```
#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;
  }
}
```

Hoisting

The idea is to avoid recomputing loop-invariant code each time through the body of a loop.

Before

```
#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/N));
  }
}
```

Loop Fusion

The idea is to combine multiple loops over the same index range into a single loop body, thereby saving the overhead of loop control. a.k.a. Jamming

Before

```
for (int i = 0; i < n; ++i) {
  C[i] = (A[i] <= B[i]) ? A[i] : B[i];
}

for (int i = 0; i < n; ++i) {
  D[i] = (A[i] <= B[i]) ? B[i] : A[i];
}
```

Ternary operator for `if-else`.

After

```
for (int i = 0; i < n; ++i) {
  C[i] = (A[i] <= B[i]) ? A[i] : B[i];
  D[i] = (A[i] <= B[i]) ? B[i] : A[i];
}
```

Eliminating Wasted Iterations

The idea is to modify loop bounds to avoid executing loop iterations over essentially empty loop bodies.

Before

```
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;
    }
  }
}
```

After

```
for (int i = 1; i < n; ++i) {
  for (int j = 0; j < i; ++j) {
      int temp = A[i][j];
      A[i][j] = A[j][i];
      A[j][i] = temp;
  }
}
```

FUNCTIONS

Inlining

The idea is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself

Before

```
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;
}
```

After

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

Inlining (2)

The idea is to avoid the overhead of a function call by replacing a call to the function with the body of the function itself

Ask the compiler to inline for you.

```
__attribute__((always_inline))
```

```
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;
}
```

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

Tail-Recursion Elimination

It is to remove the overhead of a recursive call that occurs as the last step of a function. The call is replaced with a branch to the top of the function, and the storage for the local variables of the function is reused by the erstwhile recursive call.

Before

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

After

```
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 is to increase the size of the base case and handle it with more efficient code that avoids function-call overhead.

Before

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

After

```c
#define THRESHOLD 64
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

New Bentley Rules

Data structures

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

Loops

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

Logic

- Constant folding and propagation
- Common-subexpression elimination
- Algebraic identities
- Creating a fast path
- Short-circuiting
- Ordering tests
- Combining tests

Functions

- Inlining
- Tail-recursion elimination
- Coarsening recursion

Closing Advice

- Avoid premature optimization. First, get correct working code. Then optimize, preserving correctness by regression testing.
- Reducing the work of a program does not necessarily decrease its running time, but it is a good heuristic.
- Many optimizations involve tradeoffs. Use a profiler to see what code needs to be optimized. (See Homework 2.)
- The compiler automates many low-level optimizations, but not all.

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