

CS260 – Lecture 11
Yan Gu

Algorithm Engineering (aka. How to Write Fast Code)

New Bentley rules for
modern programming

Many slides in this lecture are borrowed from the second lecture in 6.172 Performance Engineering of Software Systems at MIT. The credit is to Prof. Charles E. Leiserson, and the instructor appreciates the permission to use them in this course.

CS260:
Algorithm
Engineering
Lecture 11

Scientific writing

New Bentley rules

Trump accused Democrats of introducing the war powers resolution based on political calculations ahead of the 2020 presidential election. "This was a very insulting resolution introduced by Democrats as part of a strategy to win an election on November 3 by dividing the Republican Party," Trump said in a Wednesday statement. "The few Republicans who voted for it played right into their hands." In a statement following the president's veto, Kaine said that Trump wasn't following through on his campaign promise to cease endless wars. Democrats and some Republicans have argued that they didn't see "compelling" evidence that the U.S. faced an imminent threat posed by Iran and Soleimani, pushing back on the administration's main claim in justifying the strike against the military general. "While the president has the right to take action against truly imminent dangers ... it is our responsibility to ensure he's taking the right actions to protect Americans and our interests," Senate Foreign Relations Committee Ranking Member Bob Menendez of New Jersey said. "The president does not have authority to undertake any military action he likes. "Republican lawmakers have been more willing to buck the president when it comes to some foreign policy matters and military authority. Last March, the Senate adopted a bipartisan resolution that would scale back U.S. support for the Saudi-led military campaign in Yemen, which marked the first time the body voted to cease military support for a war in which U.S. involvement hasn't been approved by Congress. Trump also vetoed that measure and supporters in the Senate were unable to overturn it last May. On Thursday, a majority of GOP senators ultimately sided with the president and paved the wave for his broad authority on potential and future military action. "We must maintain the measure of deterrence we restored with the decisive strike on Soleimani," Senate Majority Leader Mitch McConnell of Kentucky said before the vote. "That starts today with upholding the president's rightful veto a misguided war powers resolution." Trump accused Democrats of introducing the war powers resolution based on political calculations ahead of the 2020 presidential election. "This was a very insulting resolution introduced by Democrats as part of a strategy to win an election on November 3 by dividing the Republican Party," Trump said in a Wednesday statement. "The few Republicans who voted for it played right into their hands." In a statement following the president's veto, Kaine said that Trump wasn't following through on his campaign promise to cease endless wars. Democrats and some Republicans have argued that they didn't see "compelling" evidence that the U.S.

Writing also has purposes, just like your presentations

- E.g., essays in GRE/SAT tests
- Know what your goals are, and strive the best to explain / clarify them
- Paper Reading: not teach me what this paper is about / how much effort you have spent on reading it; show your understanding of the content, the same as the presentation
- Project Proposal: describe the problems you want to solve, prior work, potential challenge, and your plan
- Project Report: More explanation later

Writing style can help a lot!

- In your talks, you use slide titles to guide the audience
- In your report / proposal / paper reading, use section titles (subsections, paragraph headers) and good paragraphing
- See the papers and sample midterm report for references

Follow the guidance!

- I know many of you do not have much experience in scientific writing

The goal of this project is not to get the fastest implementations for these problems in the world---these are some of the mostly-studied algorithms in computing for decades. The goal is to provide a chance for you to start from a simple algorithm and experience how you can optimize the performance based on the theory and practice mentioned in the class. Namely, in your report, you should provide all versions of your implementation and emphasize where the improvements are from and by how much. Of course, the running time for the final version also matters since basically that shows how much you attempt to engineer the performance. The midterm project will give you an intuition about what you can do in a few weeks so it will help you to pick a final project with more realistic goals.

Follow the guidance!

- I know many of you do not have much experience in scientific writing

Provide all versions of your implementation

Show how you engineer the performance and by how

Analysis of performance

Design

How to guarantee correctness

Explaining the optimizations

Performance

Experiment setup

Show speedup

Show scalability

Show other measures

Problem adjust (+2 for semisort /-2 for MM) / bonus

Expected outcome of this course

How to write faster code

How to speak (communicate)

How to write (scientific writing)

- **The last two aspects are crucial because:**
 - You are all very good at CS techniques, and it takes a lot to further improve
 - If you cannot communicate well, employers are hard to identify you from the great majority of other CS undergrad/grad students
 - Communication is an orthogonal dimension, and easy to improve from bad/okay to good (still hard from good to great)
- **But most courses do not cover them because they are costly**
 - Most courses have >30 students, and grading is done by TAs and readers
 - I spend ~4h for every of your talk (does not scale to larger classes)
 - You should catch the opportunity since there won't be many courses at UCR in this style

Some reminders

- **Office hour: 1:30-2:30pm Tuesday**
- **First weekly report for final project is due this Wednesday (5/13)**
- **Paper reading is due this Friday (5/15)**

CS260:
Algorithm
Engineering
Lecture 11

Scientific writing

New Bentley rules

Definition of “Work”

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

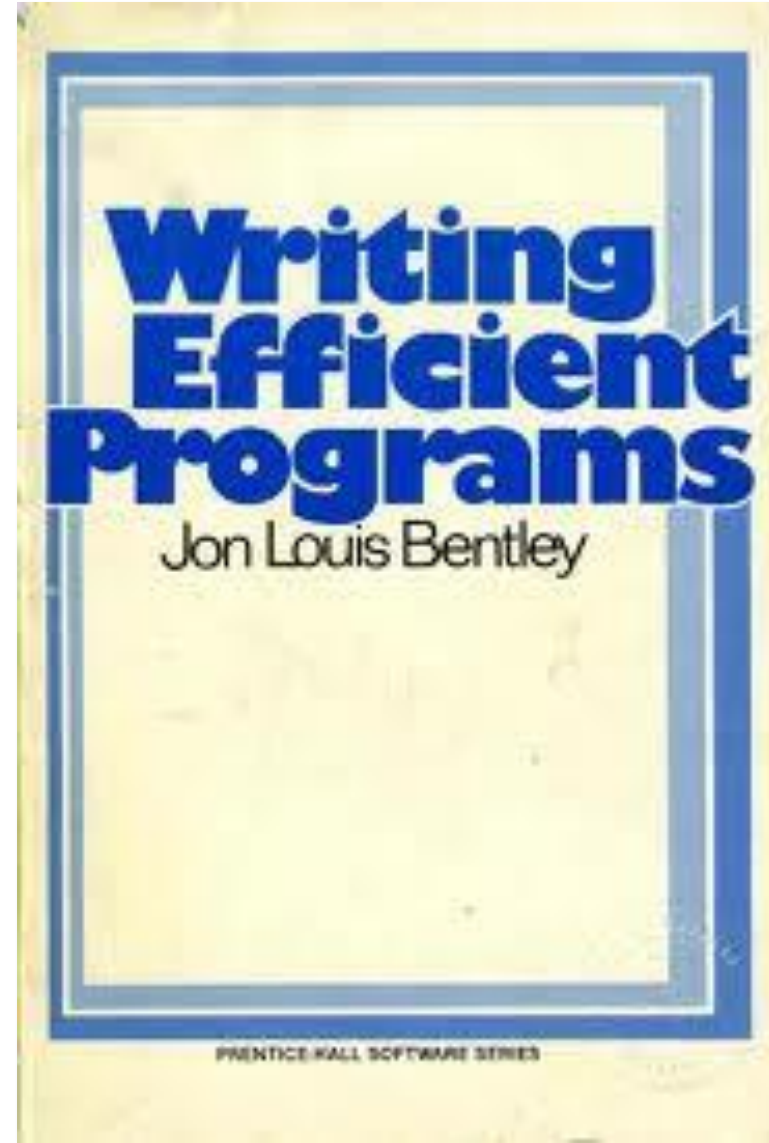


Optimizing Work

- **Algorithm design** can produce **dramatic reductions** in the amount of work it takes to solve a problem, as when a $\Theta(n \log n)$ -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 Rules

Jon Louis Bentley



1982

New “Bentley” Rules

- Most of Bentley’s original rules dealt with **work**, but some dealt with the vagaries of computer architecture four decades ago
- We have created a **new set of Bentley rules** dealing only with work
- We have discussed **architecture-dependent optimizations** in previous lectures



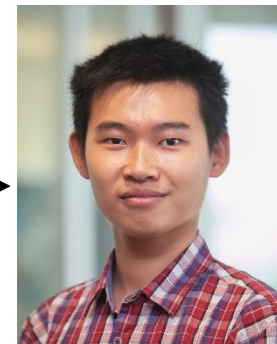
Jon Louis
Bentley



Charles
Leiserson



Guy
Blelloch



Yan
Gu

New Bentley Rules

Data structures

- Packing and encoding
- Augmentation
- Precomputation
- Compile-time initialization
- Caching
- Lazy evaluation
- 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
- Creating a fast path
- Combining tests

Functions

- Inlining
- Tail-recursion elimination
- Coarsening recursion

[link](#)

Data Structures

Packing and Encoding

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 “September 12, 2020” can be stored in 18 bytes — more than two double (64-bit) words — which must be moved whenever a date is manipulated.
- Assuming that we only store years between 4096 B.C.E. and 4096 C.E., there are about $365.25 \times 8192 \approx 3 \text{ M}$ dates, which can be encoded in $\lceil \log_2(3 \times 10^6) \rceil = 22$ bits, easily fitting in a single (32-bit) word.
- But determining the month of a date takes more work than with the string representation.

Packing and Encoding (2)

Example: Packing dates

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

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

- This packed representation still only takes 22 bits, but the individual fields can be extracted much more quickly than if we had encoded the 3M dates as sequential integers.

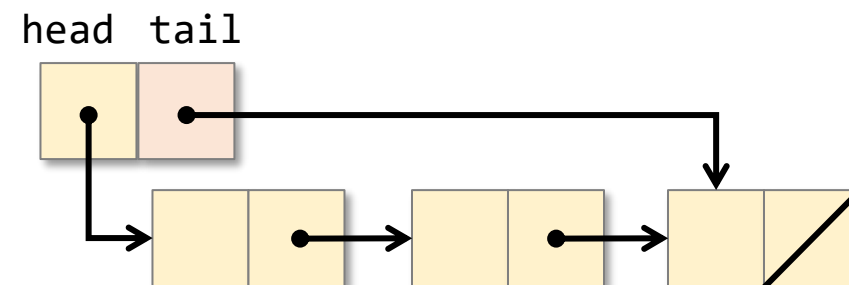
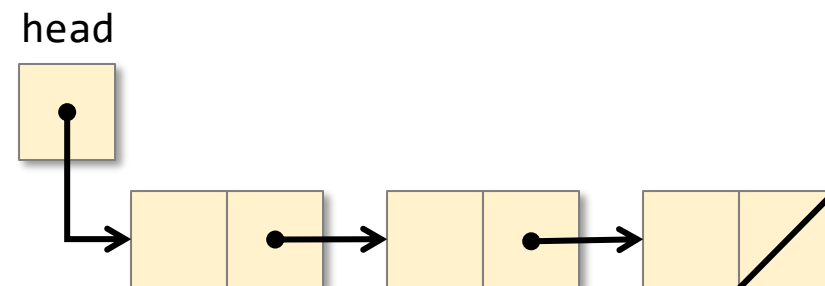
Sometimes **unpacking** and **decoding** are the optimization, depending on whether more work is involved moving the data or operating on it.

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



Precomputation

The idea of **precomputation** is to perform calculations in advance so as to avoid doing them at “mission-critical” times

Example: Binomial coefficients

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

Computing the “choose” function by implementing this formula can be expensive (lots of multiplications), and watch out for integer overflow for even modest values of **n** and **k**

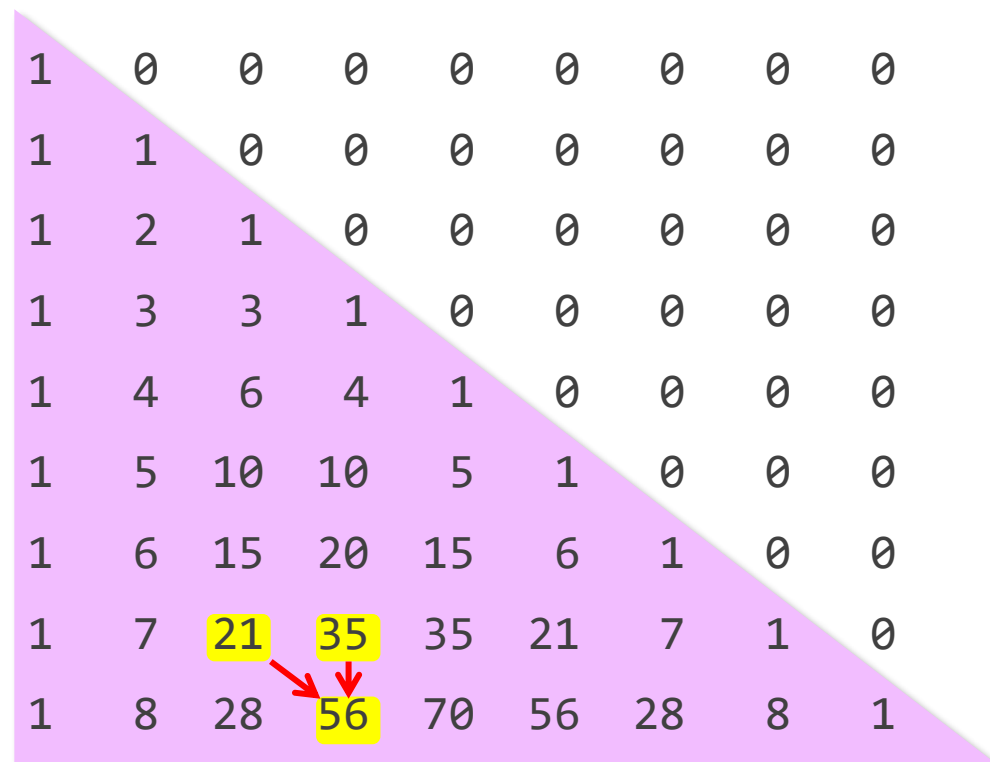
Idea: Precompute the table of coefficients when initializing, and perform table look-up at runtime

Pascal's Triangle

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

```
#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;
        }
    }
}
```



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

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

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

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

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

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

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

Sparsity (2)

Compressed Sparse Row (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

0	3	0	0	0	1	0
1	0	4	1	0	5	9
2	0	0	0	2	0	6
3	5	0	0	3	0	0
4	0	0	0	0	5	0
5	0	0	0	8	9	7
	0	1	2	3	4	5

$n = 6$
 $nnz = 14$

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

Sparsity (3)

CSR matrix-vector multiplication

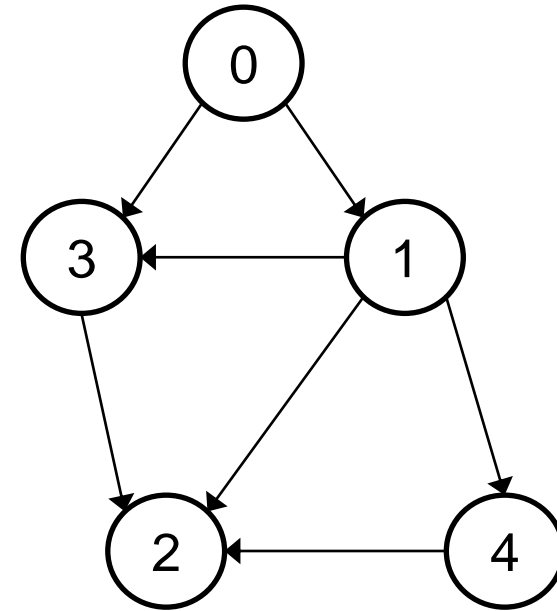
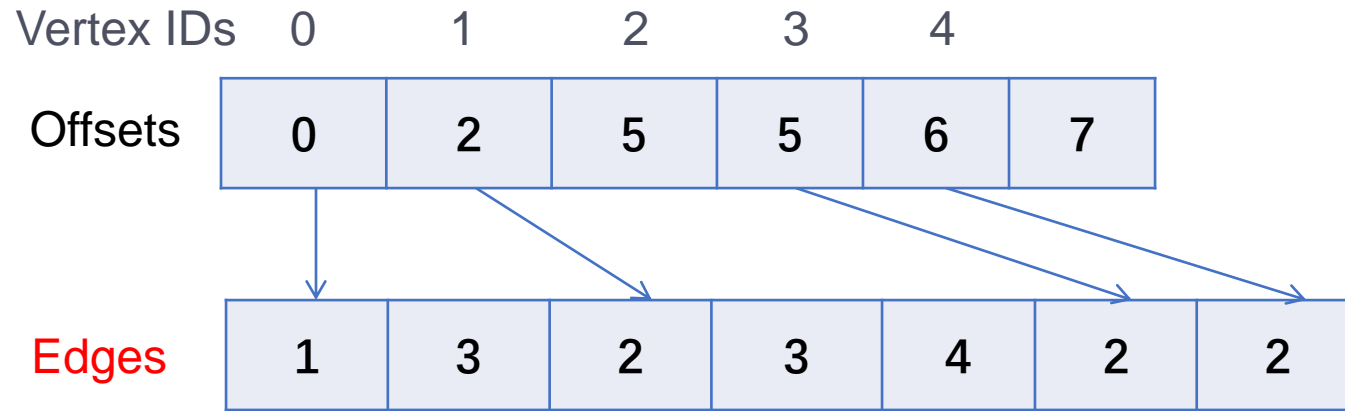
```
typedef struct {
    int n, nnz;
    int *rows;    // length n
    int *cols;    // length nnz
    double *vals; // length nnz
} sparse_matrix_t;

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

Sparsity (4)

Storing a static sparse graph



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

Can store edge weights with an additional array or interleaved with **Edges**

Logic

Constant Folding and Propagation

The idea of **constant folding and propagation** is to evaluate constant expressions and substitute the result into further expressions, all during compilation

```
#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

Algebraic Identities

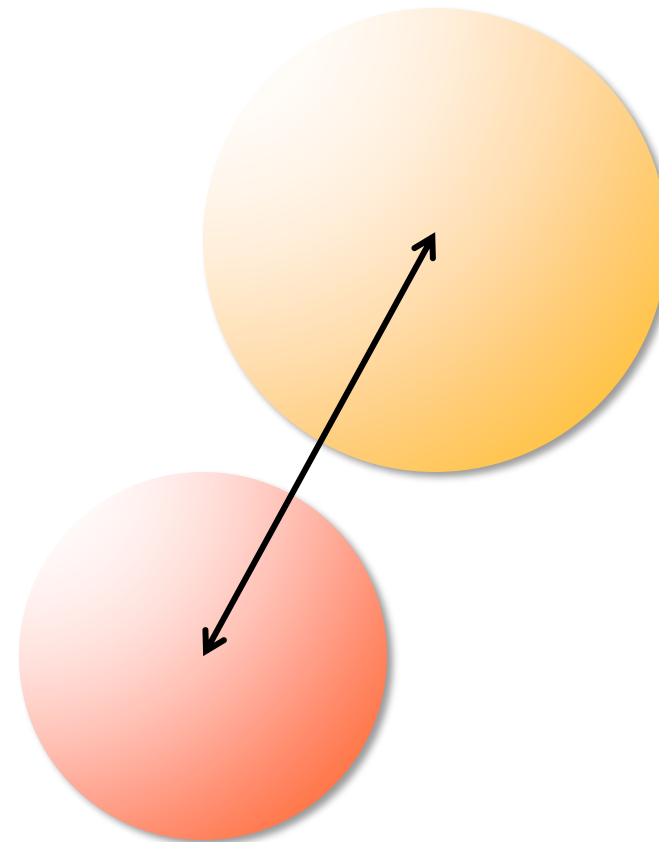
The idea of **exploiting algebraic identities** is to replace expensive algebraic expressions with algebraic equivalents that require less work.

```
#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;
}
```



Algebraic Identities

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

```
#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;
}
```

```
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);
}
```

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

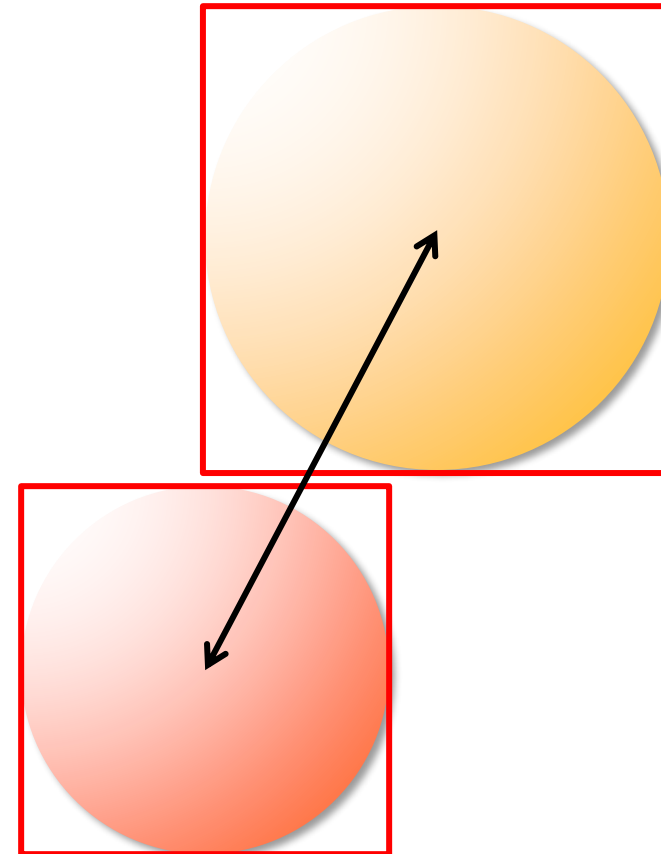
Creating a Fast Path

```
#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);
}
```



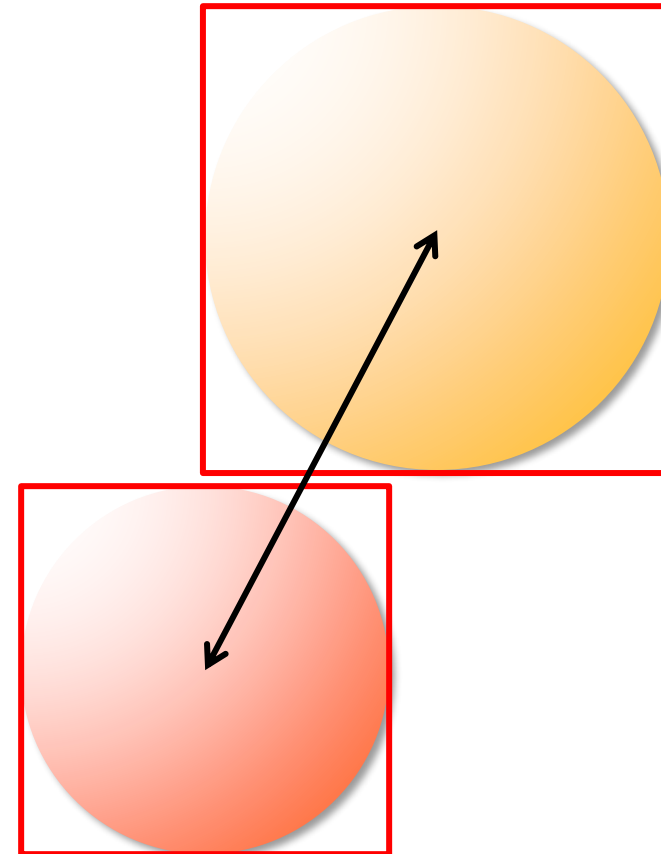
Creating a Fast Path

```
#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);
}
```



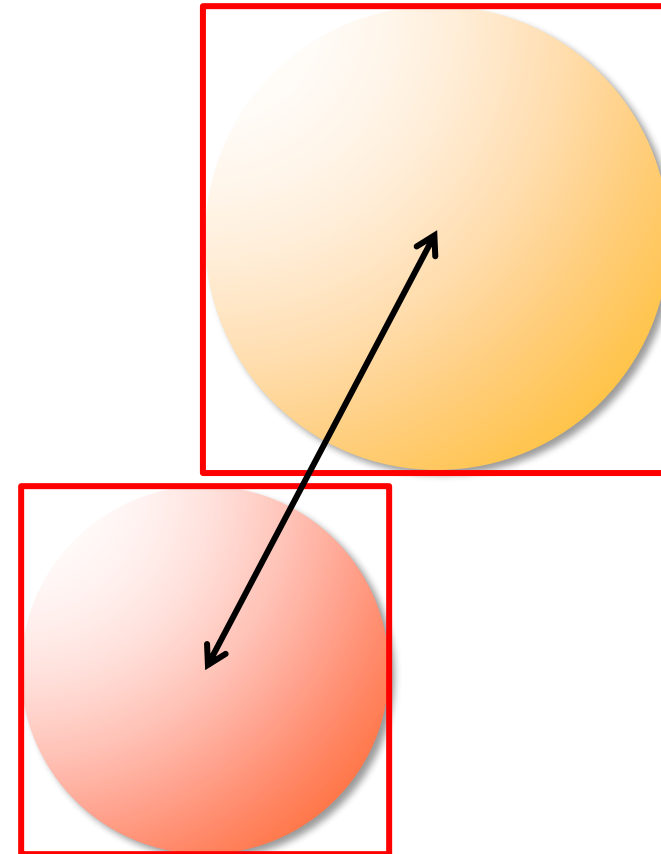
Creating a Fast Path

```
#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) {
    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

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

```
#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;
}
```

```
#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;
}
```

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

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.

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

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

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

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

```
#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;
    }
}
```

Loop Fusion

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

```
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];  
}
```

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

```
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;  
        }  
    }  
}
```

```
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 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;  
}
```

```
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

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;  
}
```

```
static 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;  
}
```

Tail-Recursion Elimination

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

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

```
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

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

```
#define THRESHOLD 20  
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
- Precomputation
- Compile-time initialization
- Caching
- Lazy evaluation
- 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
- Creating a fast path
- Combining tests

Functions

- Inlining
- Tail-recursion elimination
- Coarsening recursion

[link](#)

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**.
- The **compiler** automates many low-level optimizations.
- To tell if the compiler is actually performing a particular optimization, look at the **assembly code**.