

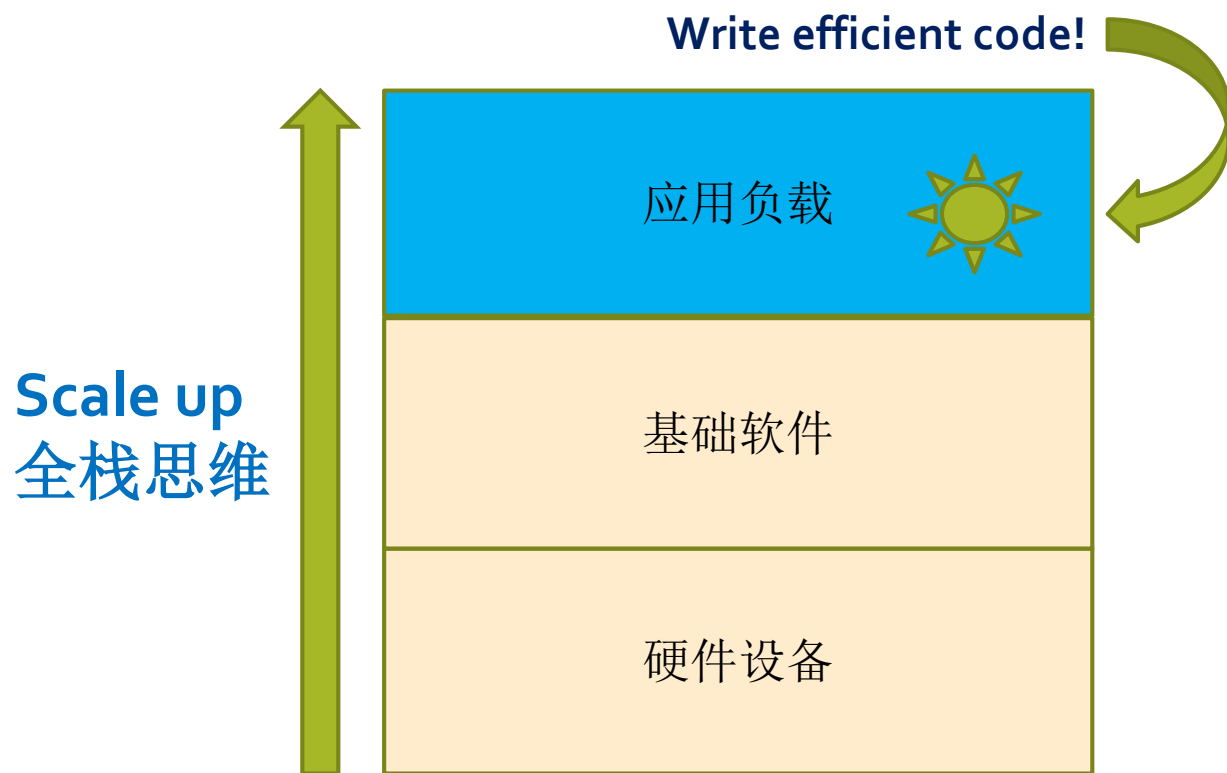


# 源程序级别的常见优化方法

黄波

bhuang@dase.ecnu.edu.cn

# 本次课的关注点



## 课程简介 + 矩阵乘法优化案例

应用负载性能测量方法  
应用负载的配置优化  
基准评测  
应用负载的性能评价

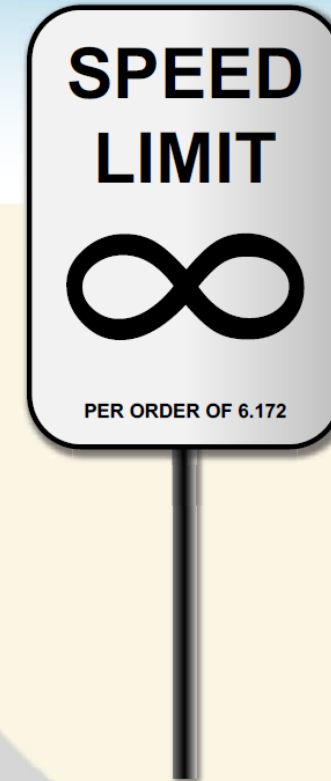
源程序级别的常见优化方法介绍  
编译器概述  
目标指令集架构及汇编语言  
C程序的汇编代码生成  
编译器的优化能力  
程序插桩及优化机会识别

计算机体系结构概论  
多核编程  
微架构性能分析方法  
高速缓存及相关优化

数据中心和云端服务优化  
机器学习框架优化简介



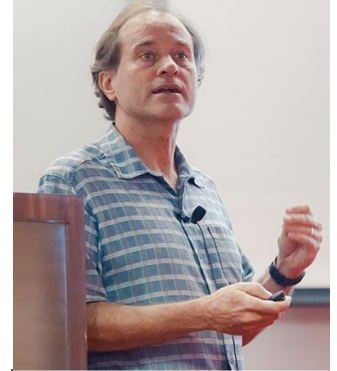
# 6.172 Performance Engineering of Software Systems



## LECTURE 2 Bentley Rules for Optimizing Work Julian Shun

© 2008–2018 by the MIT 6.172 Lecturers

1



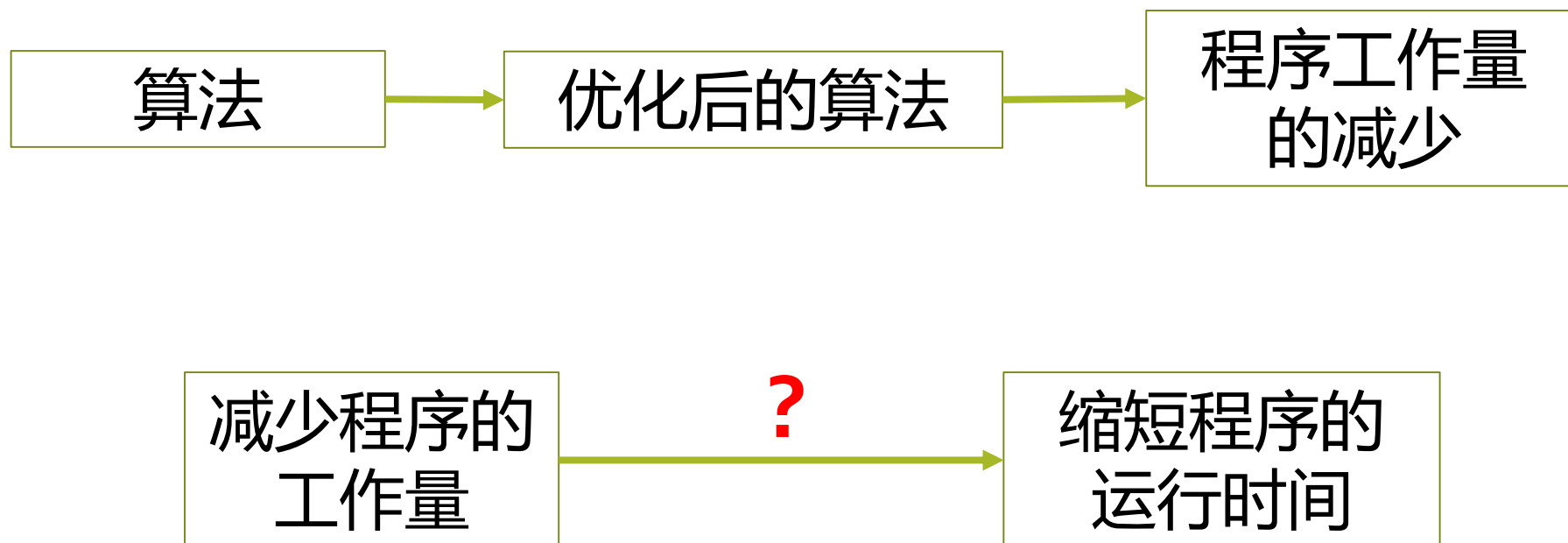
[https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6\\_172F18\\_lec2.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2018/lecture-slides/MIT6_172F18_lec2.pdf)

# 程序的工作量

**定义：** 一个程序的工作量是指在给定的输入情况下此程序运行时所执行的所有操作的总和。



# 优化



# 新宾利法则



# 新宾利法则

## 数据结构

- 打包和编码
- 数据增添
- 预先计算
- 编译时做初始化
- 缓存
- 延迟计算
- 稀疏性

## 程序逻辑

- 常数折叠与传播
- 公共子表达式消除
- 代数恒等替换
- 创建快速通道
- 逻辑短路
- 判断排序
- 组合判断

## 循环

- 循环不变量外提
- “哨兵”设置
- 循环展开
- 循环合并
- 消除无用迭代

## 函数

- 函数内联
- 尾递归消除
- 粗化递归



# 编码和打包 (1)

**数据编码** (encoding) 是指如何把数据表示成计算机程序可以理解和操作的方式。在数据结构的优化中, **打包** (packing) 是指在一个原来存储一个数值的存储空间里面存储两个或者两个以上的数值, 即相应数值对应的编码用更少的位数就可以表示。

## 示例: 日期编码

- 字符串表示: "October 12, 2022" ( **16** bytes)  
日期操作 (e.g. 取年、月、日的值, 两个日期之间的天数) ?
- 假如我们只考虑从公元前4096年到公元4096年之间的日期  
约  $365.25 \times 8192 \approx 3,000,000$  天, 22位 ( $\lceil \log_2(3000000) \rceil$ ) (**4 字节**)  
日期操作?

Q: 选择何种数据编码的原则?



# 编码和打包 (2)

## 示例: 日期的打包表示

- 用位域表示年、月、日 (22位)
- 日期操作?

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

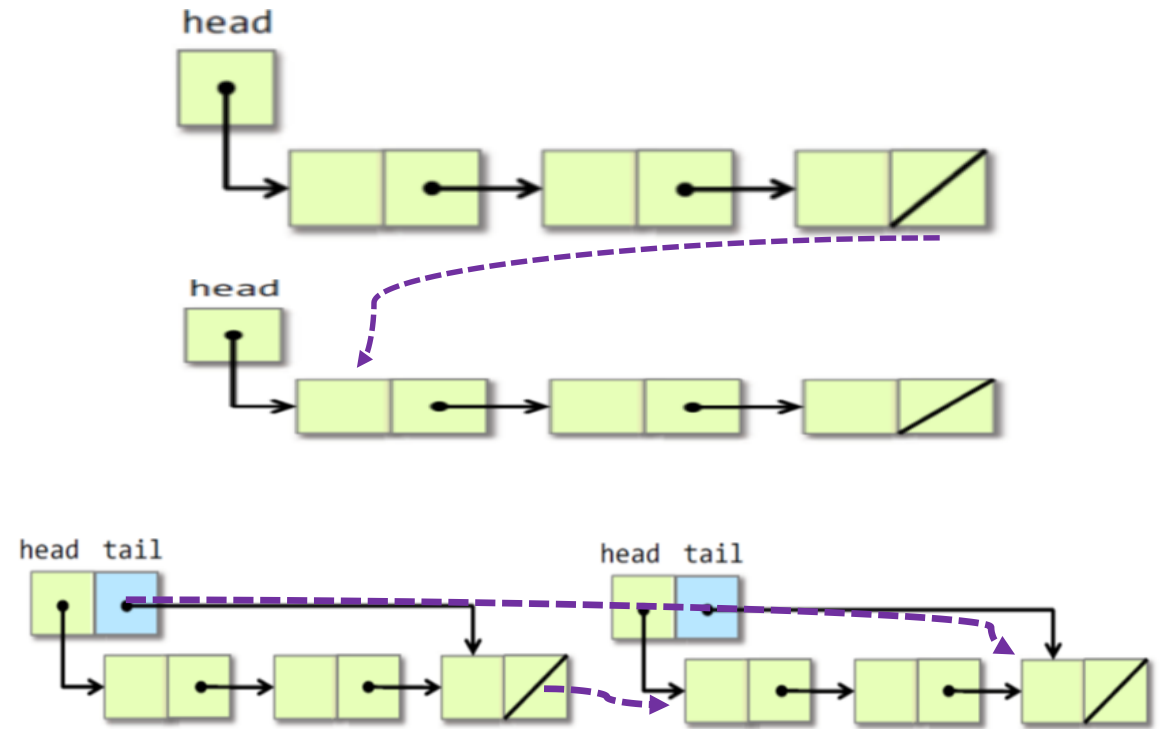
打包和解包、编码和解码，这些操作的工作量和频度都是我们设计数据结构时需要仔细考虑的问题。

# 数据增添

数据增添是指在一个数据结构中通过添加一些额外信息来减少基于此数据结构所进行的常见操作的工作量。

**示例:** 串联两个单向链表

给单向链表的数据结构增加一个尾指针，以指向单向链表的最后一个节点



# 预先计算

- **预先计算**的思想是提前进行计算以避免在程序运行的时候占用程序运行时间来执行计算。

**示例：**计算二项式系数

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

普通思路：三次求取阶乘的函数调用：  $factorial(n) / (factorial(k) * factorial(n-k))$

– 计算量大、可能会溢出

**想法：**预先计算好常用的二项式系数值，并把这些计算好的二项式系数值存放在一个数组中，然后在需要用到某个二项式系数时通过对数组元素的引用就可以取得相应的二项式系数的值。

# 帕斯卡三角形

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

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

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

$n < k$

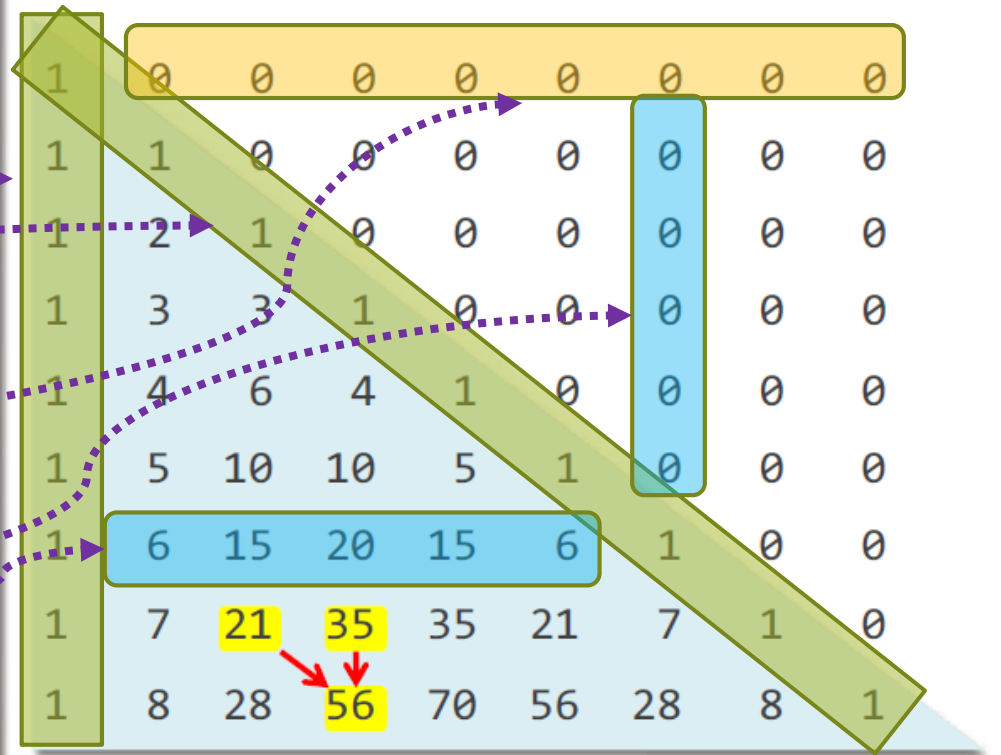
1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0
1	3	3	1	0	0	0	0	0
1	4	6	4	1	0	0	0	0
1	5	10	10	5	1	0	0	0
1	6	15	20	15	6	1	0	0
1	7	21	35	35	21	7	1	0
1	8	28	56	70	56	28	8	1

Q: 递归实现是否是最佳策略?

# 帕斯卡三角形的非递归计算

```
#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;
        }
    }
}
```



$n < 100$ 情况下的每个二项式系数值 $C(n, k)$ 的引用可以直接引用数组元素 $choose[n][k]$

# 编译时做初始化(1)

在编译的时候把一些常量存放在数据结构内，以避免在程序运行的时候去计算出那些常量。

**示例:**

```
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, },  
};
```

Q: 这个数组如何生成?



# 编译时做初始化 (2)

**思路：**元程序设计 (*metaprogramming*)

```
int main(int argc, const char *argv[]) {
    init_choose();
    printf("int choose[10][10] = {\n");
    for (int a = 0; a < 10; ++a) {
        printf("    {");
        for (int b = 0; b < 10; ++b) {
            printf("%3d, ", choose[a][b]);
        }
        printf("},\n");
    }
    printf("};\n");
}
```



# 缓存

指保存已经计算过的结果，从而在下次做同样的计算时只需访问已经保存的结果而不需要重新计算。

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



假设有2/3的概率缓存命中，可以快大约30%。

```
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;  
}
```

# 稀疏性 (1)

最快的计算就是不做任何计算。如果有些数据结构里面存储有大量的空值或者零，我们称这类数据结构具有稀疏性，或者把这类数据结构称为稀疏数据结构。对于稀疏性比较高的数据结构，一种直观的优化思路是避免对零的计算和存储。

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

采用普通的矩阵表示和相应算法，需要执行 $6 \times 6 = 36$ 次乘法，然而这36次乘法中只有14次乘法的因子不含零。

# 稀疏性 (2)

**压缩稀疏行** (*Compressed Sparse Row*, 简称 *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	4	3	4	5
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

$n = 6$   
 $nnz = 14$

0	1	2	3	4	5
---	---	---	---	---	---

存储空间:  $O(n+nnz)$  而不是  $n^2$ .

# 稀疏性 (3)

- CSR格式下的矩阵与向量相乘算法:

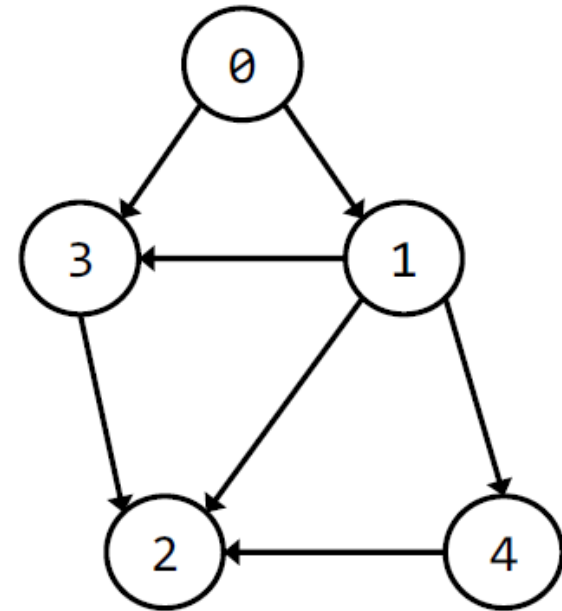
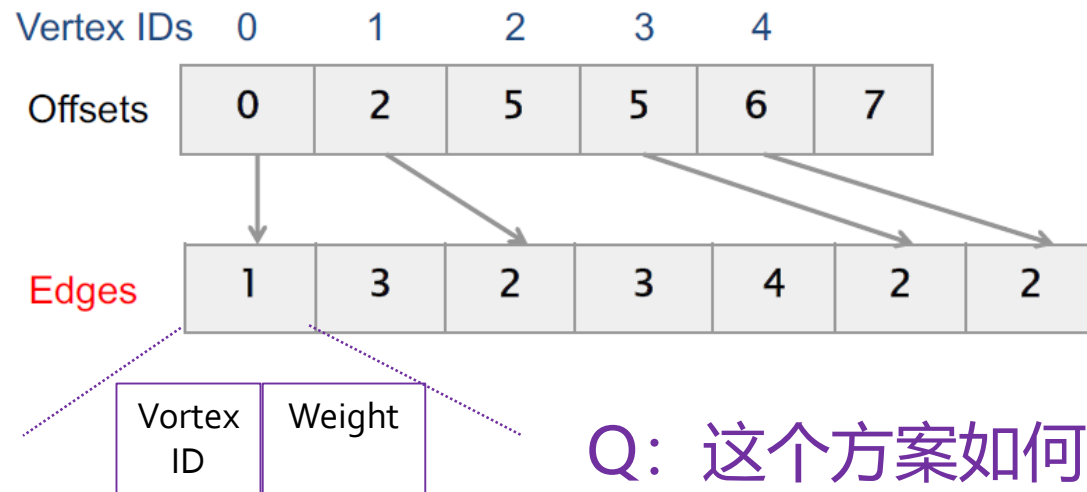
```
typedef struct {
    int n, nnz;
    int *rows;      // length n
    int *cols;      // length nnz
    double *vals;   // length nnz
} sparse_matrix_t;

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];
        }
    }
}
```

乘法次数:  $nnz$  vs.  $n^2$

# 稀疏性 (4)

## • 存储静态稀疏图



- 可以让很多图算法更加有效地执行，这些图算法包括图的宽度优先算法、PageRank算法等；
- 如果稀疏有向图的边上有权重，可以在`Offsets`和`Edges`的基础上再加一个`Weights`数组，数组`Weights`的元素个数与`Edges`的元素个数相同，不过里面存放的值是权重。

# 常数折叠与传播 (CF & CP)

在编译阶段对常数表达式进行计算，如果某个变量的值在编译阶段进行计算后确定为常数，则进一步对程序中关于这个变量的引用进行常数替代。

```
#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;
    // ...
}
```

Q：用宏来定义常量与用const来定义常量的区别？

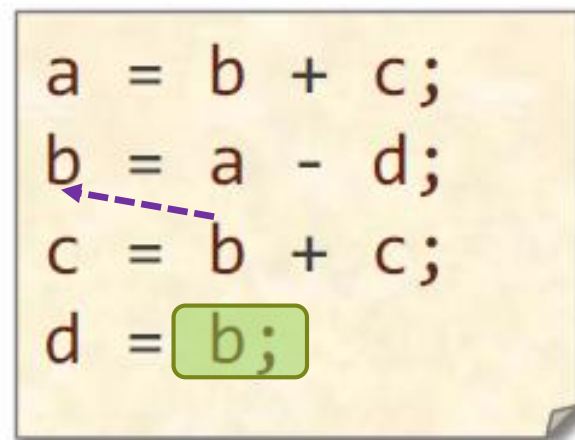


# 公共子表达式消除 (CSE)

对于程序中出现的公共子表达式（表达式相同，而且表达式的两次出现之间表达式中的变量没有被重新赋值过），为避免重复计算，在第一次对公共子表达式进行计算后把计算出来的值存储在某个变量里，以便使用这个变量来替换后续公共子表达式的出现。

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

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

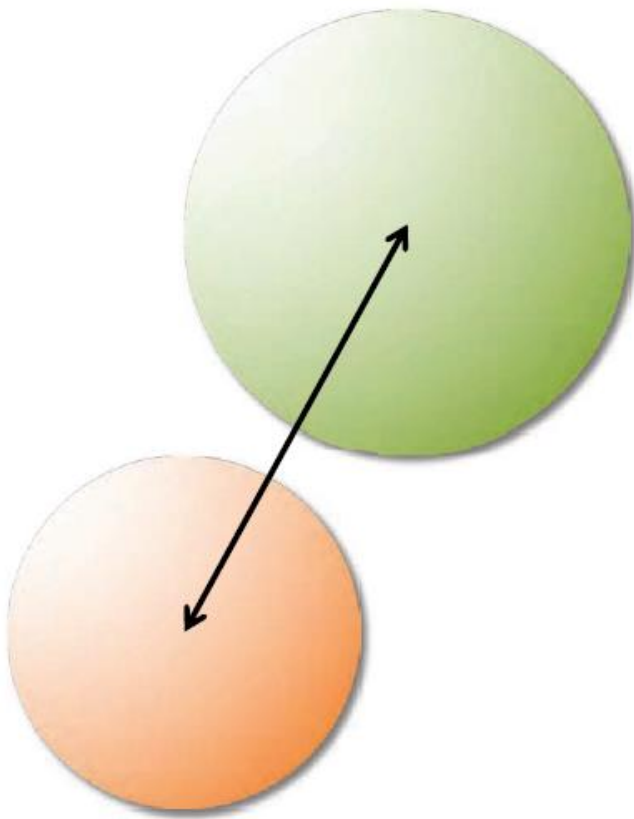


$b + c$  由于在两次出现之间存在对  $b$  的重新赋值，所以不是公共子表达式



# 代数恒等替换

开销比较大的代数表达式可以用开销相对小的等价代数表达式来替换。



```
#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;
}
```

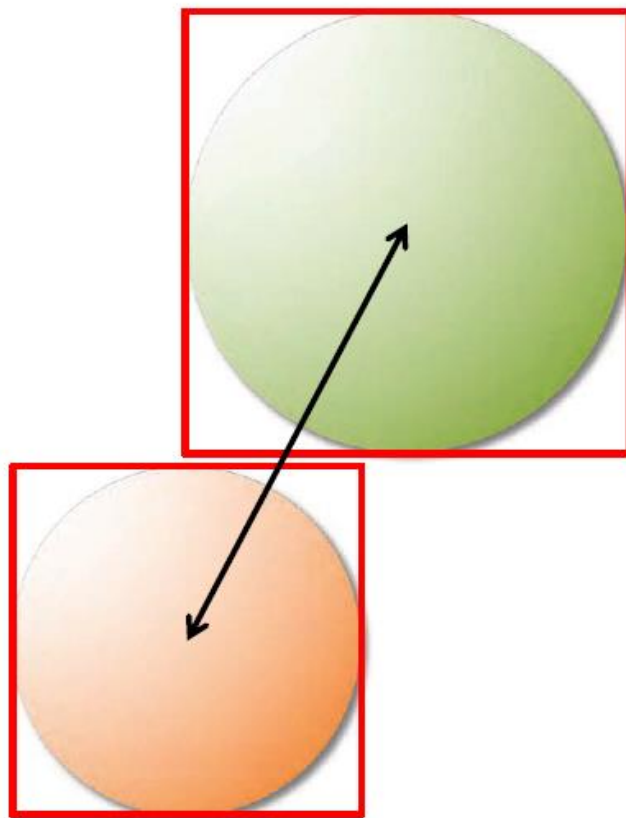
```
bool collides(ball_t *b1, ball_t *b2) {
    double d = sqrt(square(b1->x - b2->x)
                    + square(b1->y - b2->y)
                    + square(b1->z - b2->z));
    return d <= b1->r + b2->r;
}
```

```
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);
}
```

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

# 创建快速通道

如果在大多数情况下用一些简单的逻辑就可以完成判断，而无需进入更加复杂的逻辑判断，在程序逻辑中我们就可以把那些简单的逻辑判断写在前面，从而使得程序在大多数情况下不进入复杂的逻辑判断以减少工作量。



```
#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);
}
```

# 逻辑短路

在进行一系列的测试判断时，一旦测试判断的答案已经知晓，则可以马上结束测试判断，以避免浪费时间和资源做无效的工作。

```
#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;
}
```

Q: 逻辑短路后的代码一定运行时间更短么？

```
#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;
}
```


逻辑与 (&&) 和逻辑或 (||) 是常见的两个短路运算符，但位与 (&) 与位或 (|) 不是。



# 判断顺序

在进行一系列的逻辑判断时，我们可以通过重新调整每个子判断的顺序从而让整体判断的结果尽早得出，以减少计算量。

```
#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;
}
```

Q：顺序调整的依据是什么？

# 组合判断 (1)

把一系列的子判断组合替换成一个判断或者一个switch语句。

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

$a+b+c \rightarrow \text{carry, sum}$

```
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;
        }
    }
}
```

# 组合判断(2)

把一系列的子判断组合替换成一个判断或者一个switch语句。

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

Q: 对这个例子有没有更有效的方法?

```
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;
    }
}
```

# 循环不变量外提 (Hoisting, or LICM)

如果循环体内一个表达式的计算结果不会根据循环迭代的次数变化而变化，则这个表达式对于这个循环来说可以看成常量，它的计算可以被提到循环体之外，从而避免每次循环迭代都对此表达式进行重复计算。

```
#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;
    }
}
```



# “哨兵”设置

为了简化边界条件的判断逻辑，特别是循环出口的测试，有时需要在数据结构内设置一些特殊的“假值”（dummy values），这些特殊的“假值”被称为“哨兵”。

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

bool overflow (int64_t *A, size_t n) {
    // All elements of A are nonnegative
    int64_t sum = 0;
    for ( size_t i = 0; i < n; ++i ) {
        sum += A[i];
        if ( sum < A[i] ) return true;
    }
    return false;
}
```



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

// Assumes that A[n] and A[n+1] exist and
// can be clobbered
bool overflow(int64_t *A, size_t n) {
    // All elements of A are nonnegative
    A[n] = INT64_MAX;
    A[n+1] = 1; // or any positive number
    size_t i = 0;
    int64_t sum = A[0];
    while ( sum >= A[i] ) {
        sum += A[++i];
    }
    if ( i < n ) return true;
    return false;
}
```

Why?

# 循环展开 (Loop Unrolling)

循环展开试图把连续的若干个循环迭代变成一个更大的循环迭代，从而减少总的循环迭代次数，并相应地**减少了循环控制指令的执行次数**。循环展开通常分成以下两类：

- **完全循环展开**：所有的循环迭代都被展开，整个循环变成了顺序执行的语句序列。
- **部分循环展开**：只有若干个（不是全部）循环迭代被展开，部分循环展开后循环体变大，循环迭代次数减少。

(a) 完全循环展开

```
int sum=0;
for (int i = 0; i < 4; ++i ) {
    sum += A[i];
}
```



```
int sum=0;
sum += A[0];
sum += A[1];
sum += A[2];
sum += A[3];
```

(b) 部分循环展开

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



```
int sum=0, 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];
}
```


Q：循环展开还有其它什么好处？  
又可能有哪些弊端？

# 循环合并 (Loop Fusion/Jamming)

把两个或者多个具有相同循环迭代次数的循环合并成一个循环，以减小循环控制的开销。

```
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];  
}
```

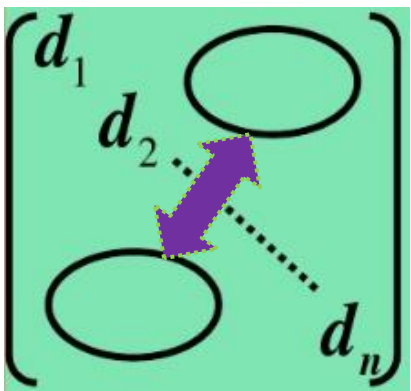
```
for (int i = 0; i < n; ++i) {  
    C[i] = (A[i] <= B[i]) ? A[i] : B[i];  
    D[i] = (A[i] <= B[i]) ? B[i] : A[i];  
}
```



Q: 循环合并有何优缺点?

# 消除无用迭代

消除无用迭代的思想是修改循环的边界条件以避免执行那些实际上循环体为空的循环迭代。



```
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;  
    }  
}
```

# 函数内联 (1)

函数内联的思想是通过把函数调用替代成被调用的函数体，从而避免了函数调用的开销。

```
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;  
}
```

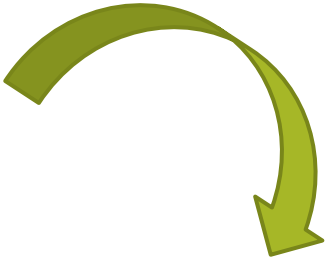
Q: 函数内联有什么副作用?



## 函数内联 (2)

函数内联的思想是通过把函数调用替代成被调用的函数体，从而避免了函数调用的开销。

```
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;  
}
```


Q: 函数内联与宏定义有何异同点?

# 尾递归消除

尾递归是一类比较特殊的递归函数调用，这类递归调用出现在函数尾部，是当前递归函数执行时的最后一条语句。

尾递归函数的调用可以转换成语义等价的循环语句。

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



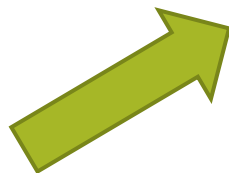
```
void quicksort(int *A, int n) {  
    while (n > 1) {  
        int r = partition(A, n);  
        quicksort (A, r);  
        A += r + 1;  
        n -= r + 1;  
    }  
}
```



# 粗化递归(coarsening recursion)

- 假如一个功能的实现有递归算法和非递归算法，并且非递归算法在某些情况下的效率比递归算法好，粗化递归的思想是结合二者的优点，在递归处理占优的时候用递归算法，在递归处理不占优的时候用非递归算法。

```
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 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;  
    }  
}
```

# 新宾利法则 – 总结

## 数据结构

- 打包和编码
- 数据增添
- 预先计算
- 编译时做初始化
- 缓存
- 延迟计算
- 稀疏性

## 程序逻辑

- 常数折叠与传播
- 公共子表达式消除
- 代数恒等替换
- 创建快速通道
- 逻辑短路
- 判断排序
- 组合判断

## 循环

- 循环不变量外提
- “哨兵”设置
- 循环展开
- 循环合并
- 消除无用迭代

## 函数

- 函数内联
- 尾递归消除
- 粗化递归

# 本次课程总结

软件开发:

保证程序正  
确性



代码  
优化

代码  
优化:

减少  
工作量

不一定



缩短程序  
运行时间

编译器

研究和理解编译  
器生成的代码

欲知后事如何，且听下回分解 😊

