Data structures & logic optimizations

Petr Kurapov

Fall 2024

SW common optimizations walkthrough

Optimizations:

- Data structures
- Logic
- Loop level
- Function level

Saving “work” not always result in better performance

Arch independent (almost…)

Data structures

Packing – describe data with less machine memory

- Date example:
  - As a string “21 April, 2020” – 14 bytes…

Data structures

Packing

- Date example:
  - As a string “21 April, 2020” – 14 bytes…
  - As integer (0 – 4096y) * 365d ≈ 1.5M dates – requires 21 bit – fits into int32. But…

```
// Simplified just a little-bit...
int get_month(int date) {
    return (date / 30) % 12;
}
```

Data structures

Packing

- Date example:
    - As a string “21 April, 2020” – 14 bytes…
    - As integer (0 – 4096y) * 365d ≈ 1.5M dates – requires 21 bit – fits into int32.
    - As a structure, no access overhead

```
struct Date {
    int year;
    int month;
    int day;
};

int get_month2(Date &date) {
    return date.month;
}
```

Data structures

Packing

- Date example:
  - As a string “21 April, 2020” – 14 bytes…
  - As integer (0 – 4096y) * 365d ≈ 1.5M dates – requires 21 bit – fits into int32.
  - As a structure, no access overhead

```
struct Date {
    int year;
    int month;
    int day;
};
int get_month2(Date &date)
    return date.month;
mov eax, dword ptr [rdi + 4]
ret
```

Data structures

Packing

- Date example:
    - As a string “21 April, 2020” – 14 bytes…
    - As integer (0 – 4096y) * 365d ≈ 1.5M dates – requires 21 bit – fits into int32.
    - As a structure, no access overhead

```
struct Date {
    int year: 12;
    int month: 4;
    int day: 5;
};
int get_month2(Date &date)
    return date.month;
movsx   eax, word ptr [rdi]
sar     eax, 12
ret
```

Data structures

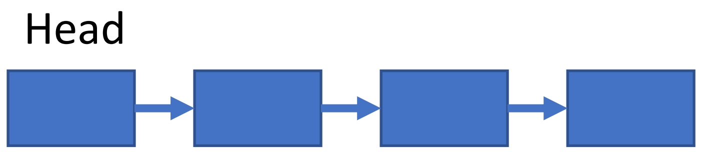
Packing also includes:

- Data structures alignment
    - Cross-page alignment: TLB misses, page faults
- Data alignment
    - Cache line aligned allocation
- Structure padding
    - Individual data items usually allocated on aligned boundaries
    - Insert additional unnamed data fields
        - Space – time tradeoff; Example int & 1 byte (+3 bytes padding) – save memory/higher access time

Data structures

Augmentation

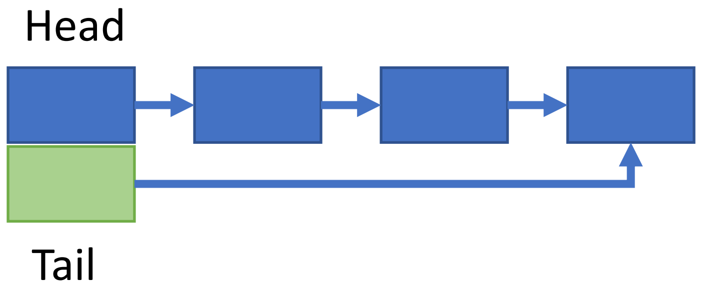
- Simple example: append one linked list to another



Data structures

Augmentation

- Simple example: append one linked list to another
- Preserve tail pointer



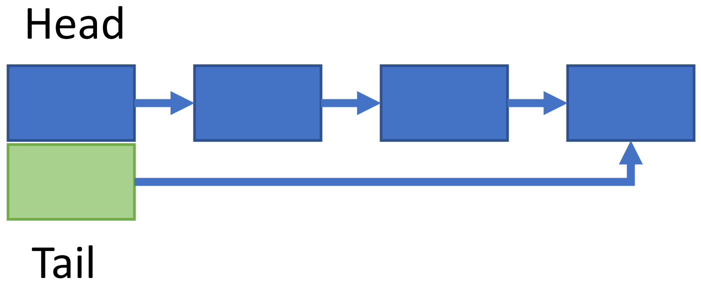
Precomputations

- Example:
  - binomial coefficients (C_n^k)

Data structures

Augmentation

- Simple example: append one linked list to another
- Preserve tail pointer



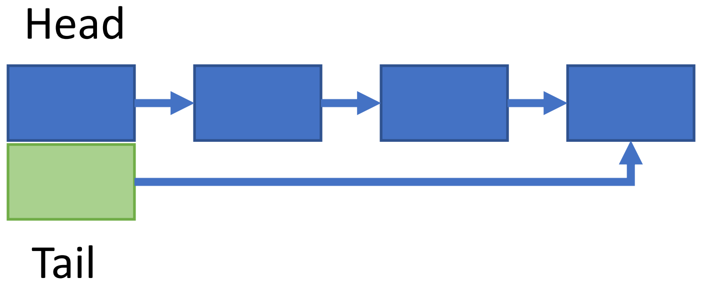
Precomputations

- Example:
    - binomial coefficients (C_n^k)
- Table lookup & compile-time initialization
- How do you precompute?

Data structures

Augmentation

- Simple example: append one linked list to another
- Preserve tail pointer



Precomputations

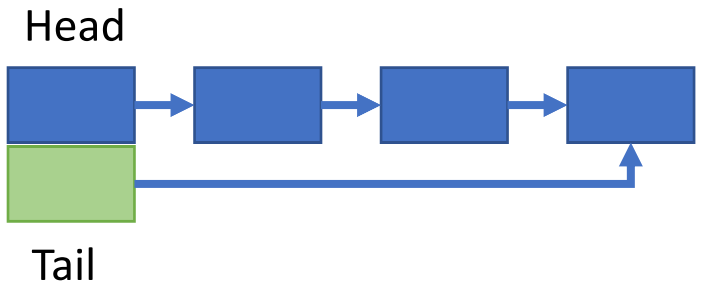
- Example:
  - binomial coefficients (C_n^k)
- How do you precompute?

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Data structures

Augmentation

- Simple example: append one linked list to another
- Preserve tail pointer



Precomputations

- Example:
  - binomial coefficients (C_n^k)
- How do you precompute?

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Data structures

What else could you do?

Precomputations

- Example:
  - binomial coefficients (C_n^k)
- How do you precompute?

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Data structures

Caching – don’t execute same thing twice

- Shouldn’t work well for this example
    - Page cache
    - Web pages, images
    - Memoization (i.e. dynamic programming algorithms design)

SW caching is a tradeoff – additional checks for cache hit

Precomputations

- Example:
    - binomial coefficients (C_n^k)
- How do you precompute?

1							
1	1						
1	2	1					
1	3	3	1				
1	4	6	4	1			
1	5	10	10	5	1		
1	6	15	20	15	6	1	
1	7	21	35	35	21	7	1

Data structures

Sparsity – avoid "zero" computations

- Naïve matrix multiplication requires Θ(n^2) operations

$$M = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 7 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 4 \\ 5 \\ 3 \end{bmatrix}$$

Only 7 multiplications are necessary out of 25.

- When do we have benefit of compression?

Data structures

Sparsity – avoid "zero" computations

- Naïve matrix multiplication requires $\Theta(n^2)$ operations

$$M = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 7 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 4 \\ 5 \\ 3 \end{bmatrix}$$

Only 7 multiplications are necessary out of 25.

- When do we have benefit of compression?
  - $nnz\ (num\ of\ nonzero) < \dfrac{m(n-1)-1}{2}$

Data structures

Sparsity

$$M = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 7 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 4 \\ 5 \\ 3 \end{bmatrix}$$

Storage complexity?

Compressed Sparse Row (CSR)

rows	0	1	2	3	5		
cols	0	2	4	0	3	1	4
vals	2	8	6	7	3	4	1

```
struct Sparse {
    int n, nnz;
    int* rows;
    int* cols; int* vals;
};
```

Data structures

Sparsity

$$M = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \\ 7 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 4 \\ 5 \\ 3 \end{bmatrix}$$

Storage complexity? O(n + nnz)

Compressed Sparse Row (CSR)

rows	0	1	2	3	5		
cols	0	2	4	0	3	1	4
vals	2	8	6	7	3	4	1

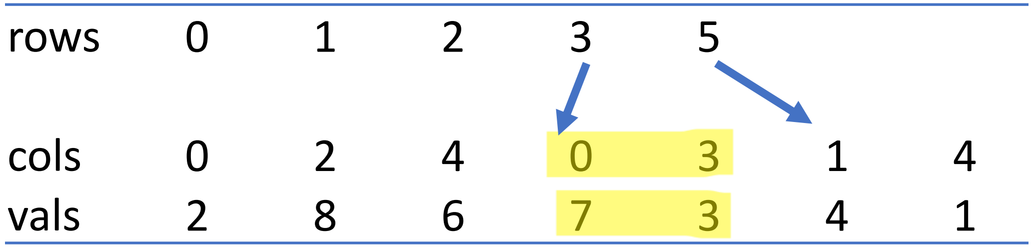
```
struct Sparse {
    int n, nnz;
    int* rows;
    int* cols; int* vals;
};
```

Data structures

Sparsity (Matrix-vector mul)

```
void mul(Sparse* A, int* x, int* M)
{
for (i = 0...A->n)
  for (k = A->rows[i]...
            A->rows[i+1]) {
    int j = A->cols[k];
    M[i] += A->vals[k] * x[j];
  }
}
```

Compressed Sparse Row (CSR)



rows	0	1	2	3	5		
cols	0	2	4	0	3	1	4
vals	2	8	6	7	3	4	1

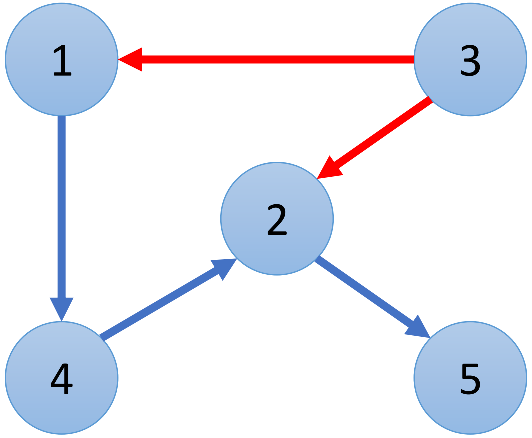
```
struct Sparse {
    int n, nnz;
    int* rows;
    int* cols; int* vals;
};
```

Data structures: graph representation recap

- Adjacency matrix

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

- $G = (V, E)$



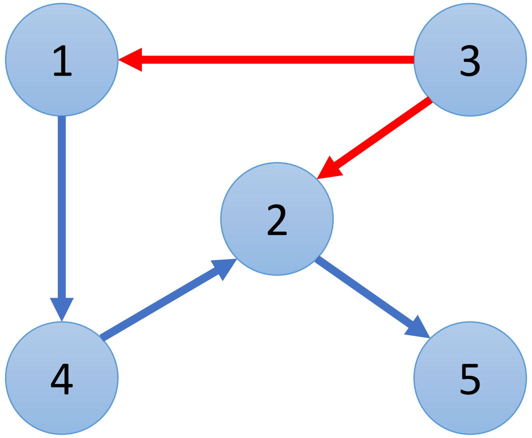
Operation	Complexity
Remove edge	
Query edge	
Add vertex	
Total space	
Get adjacent vertices	

Data structures: graph representation recap

- Adjacency matrix

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

- $G = (V, E)$

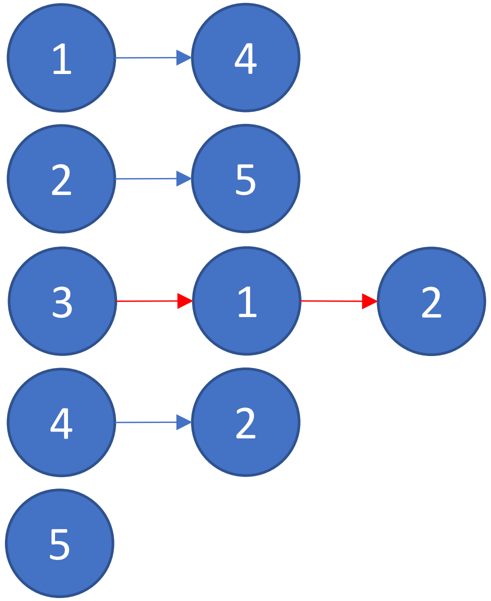


Operation	Complexity
Remove edge	$O(1)$
Query edge	$O(1)$
Add vertex	$O(\|V\|^2)^*$
Total space	$O(\|V\|^2)$
Get adjacent vertices	$O(\|V\|)$

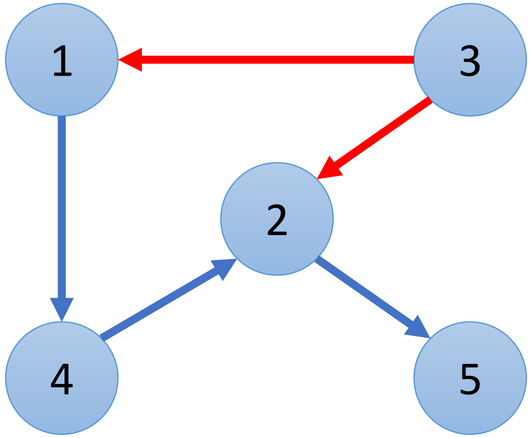
Resize/copy on add/remove vertices

Data structures: graph representation recap

- Adjacency list



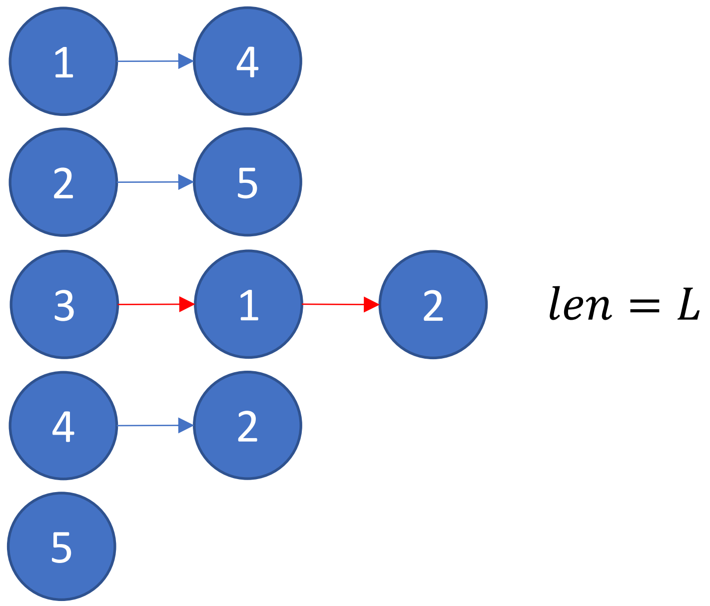
- $G = (V, E)$



Operation	Complexity
Remove edge	
Query edge	
Add vertex	
Total space	
Get adjacent vertices	

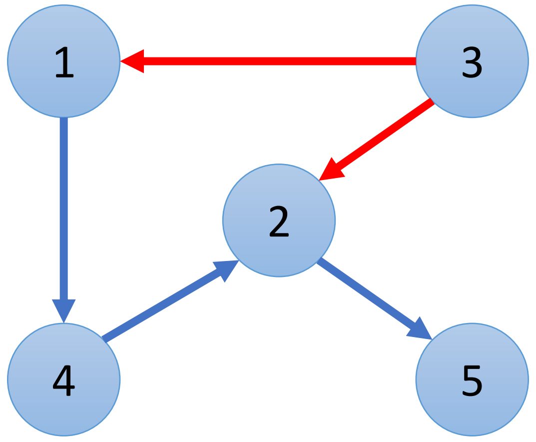
Data structures: graph representation recap

- Adjacency list



List search on add/remove vertices & edges

- $G = (V, E)$

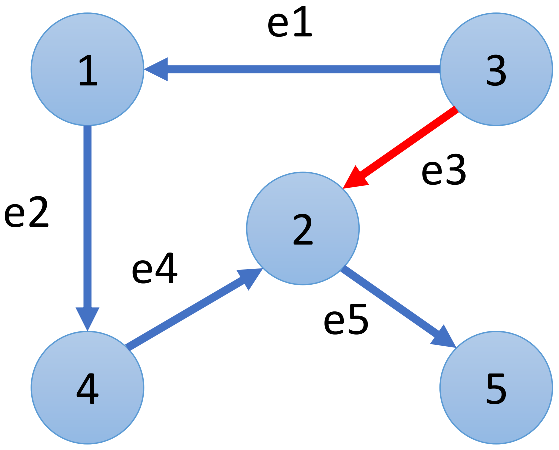


Operation	Complexity
Remove edge	$O(L)$
Query edge	$O(L)$
Add vertex	$O(1)$
Total space	$O(\lvert V\rvert + \lvert E\rvert)$
Get adjacent vertices	$O(L)$

Data structures: graph representation recap

- Incidence matrix

$$\begin{array}{c|ccccc} & e1 & e2 & e3 & e4 & e5 \\ \hline 1 & -1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & -1 & -1 & 1 \\ 3 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & -1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & -1 \end{array}$$

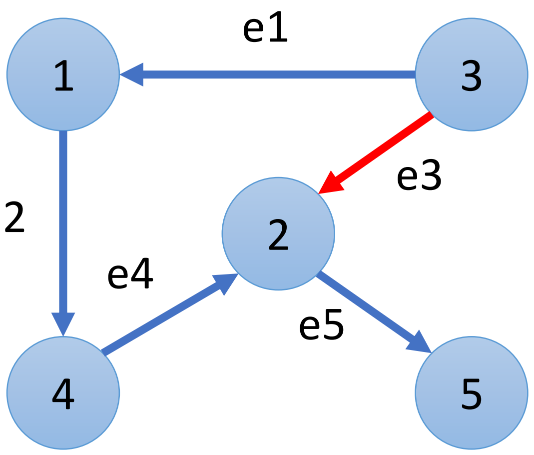


Operation	Complexity
Remove edge	
Query edge	
Add vertex	
Total space	
Get adjacent vertices	

Data structures: graph representation recap

- Incidence matrix

$$\begin{array}{c|ccccc} & e1 & e2 & e3 & e4 & e5 \\ \hline 1 & -1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & -1 & -1 & 1 \\ 3 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & -1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & -1 \end{array}$$



Operation	Complexity
Remove edge	$O(\|V\|\|E\|)^*$
Query edge	$O(1)$
Add vertex	$O(\|V\|\|E\|)^*$
Total space	$O(\|V\|\|E\|)$
Get adjacent vertices	$O(\|E\|)$

Resize/copy on add/remove vertices & edges

Data structures

Sparsity

- Can we use a similar structure for graph representation?
    - **How do you define density?**

Data structures

Sparsity

- Can we use a similar structure for graph representation?
    - How do you define density?
    - $Density = \frac{|E|}{2C_{|V|}^{2}}$ (directed)

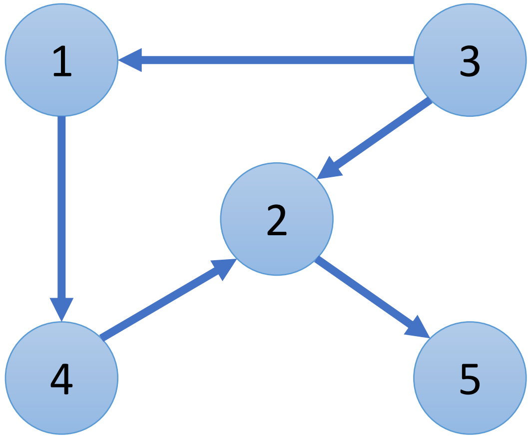
Data structures

Sparsity

- Can we use a similar structure for graph representation? Yes!
  - How do you define density?
  - $Density = \frac{|E|}{2C_{|V|}^{2}}$ (directed)

Sparse graph representation

Vertex	1	2	3	4	5
Offset	0	1	2	4	5
Edge	4	5	1	2	2



Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - Constant folding
  - Constant propagation
  - Common-subexpression elimination (CSE)
  - Loop invariant code motion (LICM)
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - **Constant folding**
  - **Constant propagation**
  - Common-subexpression elimination (CSE)
  - Loop invariant code motion (LICM)
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

```
int a = 30;
int b = 9 - (a / 5);
int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
}
return c * (60 / a);
```

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - **Constant folding**
  - **Constant propagation**
  - Common-subexpression elimination (CSE)
  - Loop invariant code motion (LICM)
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

```
int a = 30;
int b = 3;
int c;

c = b * 4;
if (c > 10) {
    c = c - 10;
}
return c * 2;
```

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
    - **Constant folding**
    - **Constant propagation**
    - Common-subexpression elimination (CSE)
    - Loop invariant code motion (LICM)
    - Loop unrolling, fusion
    - Inlining
    - Tail-recursion elimination

```
int a = 30;
int b = 3;
int c;

c = 12;
if (true) {
    c = 2;
}
return c * 2;
```

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - Constant folding
  - Constant propagation
  - **Common-subexpression elimination (CSE)**
  - Loop invariant code motion (LICM)
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

```
A = B + C
B = A − D
C = B + C
D = A − D
```

```
A = B + C
B = A − D
C = B + C
```

D = B

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - Constant folding
  - Constant propagation
  - Common-subexpression elimination (CSE)
  - **Loop invariant code motion**
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

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

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - Constant folding
  - Constant propagation
  - Common-subexpression elimination (CSE)
  - **Loop invariant code motion**
  - Loop unrolling, fusion
  - Inlining
  - Tail-recursion elimination

```
int i = 0;
if (i < n) {
    x = y + z;
    int const t1 = x * x;
    do {
        a[i] = 6 * i + t1;
        ++i;
    } while (i < n);
}
```

Logic optimizations

- Many arch independent optimizations are handled by compiler, i.e.:
  - Constant folding
  - Constant propagation
  - Common-subexpression elimination (CSE)
  - Loop invariant code motion
  - **Loop unrolling, fusion**
  - **Inlining**
  - **Tail-recursion elimination**

What do these save?

Logic optimizations

Fast path

- Use sentinels for edge cases efficient handling

Logic optimizations

Fast path

- Use sentinels or guard values for edge cases efficient handling

```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += A[i];
    if (sum < A[i])
     return false;// overflow
}
```

UB police: hands up!

Logic optimizations

Fast path

```
overflow(int*, int): # @overflow(int*, int)
        mov     al, 1
        test    esi, esi
        jle     .LBB5_5
        mov     ecx, esi
        xor     edx, edx
        xor     esi, esi
.LBB5_2: # =>This Inner Loop Header: Depth=1
        test    esi, esi
        js      .LBB5_3
        add     esi, dword ptr [rdi + 4*rdx]
        add     rdx, 1
        cmp     rcx, rdx
        jne     .LBB5_2
.LBB5_5:
        ret
.LBB5_3:
        xor     eax, eax
        ret
```

https://godbolt.org/

```
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += A[i];
    if (sum < A[i])
     return false;// overflow
}
```

Logic optimizations

Fast path

```
overflow2(int*, int): # @overflow2(int*, int)
        mov     ecx, dword ptr [rdi]
        movsxd  rax, esi
        movabs  rdx, 6442450943
        mov     qword ptr [rdi + 4*rax], rdx
        xor     eax, eax
        cmp     ecx, dword ptr [rdi]
        jl      .LBB6_3
        xor     eax, eax
.LBB6_2:  # =>This Inner Loop Header: Depth=1
        mov     edx, dword ptr [rdi + 4*rax + 4]
        add     edx, ecx
        add     rax, 1
        test    ecx, ecx
        mov     ecx, edx
        jns     .LBB6_2
.LBB6_3:
        cmp     eax, esi
        setge   al
        ret
```

```
int sum = A[0], i = 0;
A[N] = numeric_limits<int>::max();
A[N+1] = 1;
while (sum >= A[i]) {
    sum += A[++i];
}
if (i < N) return false;
```

Logic optimizations

Fast path

```
overflow2(int*, int): # @overflow2(int*, int)
        mov     ecx, dword ptr [rdi]
        movsxd  rax, esi
        movabs  rdx, 6442450943
        mov     qword ptr [rdi + 4*rax], rdx
        xor     eax, eax
        cmp     ecx, dword ptr [rdi]
        jl      .LBB6_3
        xor     eax, eax
.LBB6_2:  # =>This Inner Loop Header: Depth=1
        mov     edx, dword ptr [rdi + 4*rax + 4]
        add     edx, ecx
        add     rax, 1
        test    ecx, ecx
        mov     ecx, edx
        jns     .LBB6_2
.LBB6_3:
        cmp     eax, esi
        setge   al
        ret
```

```
int sum = A[0], i = 0;
A[N] = numeric_limits<int>::max();
A[N+1] = 1;
while (sum >= A[i]) {
    sum += A[++i];
}
if (i < N) return false;
```

Logic optimizations

Fast path

- Exploit lazy computations (probability)

  `if (cond1 && cond2 || cond3)`

- Exploit heuristics of easy-to-use conditions to shortcut expensive computations
  - Ray tracing example

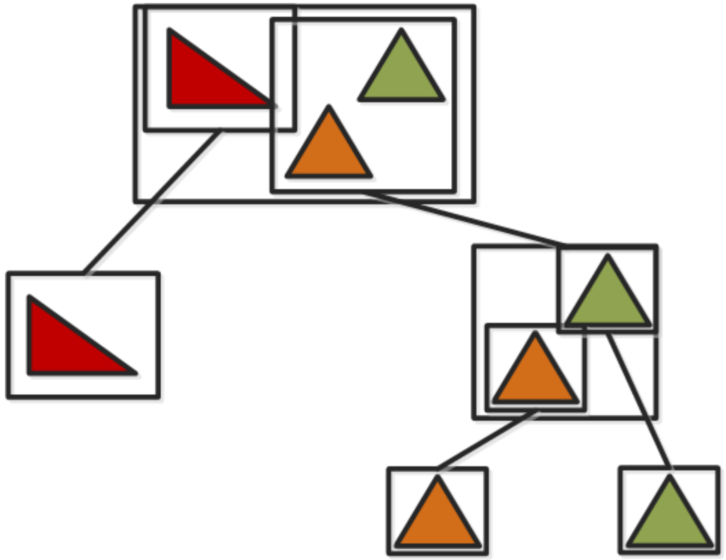
Logic optimizations

Fast path

- Exploit lazy computations (probability)

  `if (cond1 && cond2 || cond3)`

- Exploit heuristics of easy-to-use conditions to shortcut expensive computations
  - Ray tracing example
  - BVH
  - Use equivalent comparisons
    - $a < \sqrt{b}$
    - $a^2 < b$



Source: https://www.andrew.cmu.edu/user/chenxil/15418project/finalreport.html

Resources

- [1] Introduction to Algorithms, Thomas H. Cormen, chapters 22
- [2] MIT 6.172

BACKUP

Tail recursion

```
int fact(int n) {
    if (n < 2)
        return 1;
    return n*fact(n-1);
}
```

```
int f(int n, int a) {
    if (n < 2)
        return a;
    return ft(n-1, a*n);
}
```

Tail recursion

```
fact(int):
        mov     eax, 1
        cmp     edi, 1
        jle     .L1
.L2:
        mov     edx, edi
        sub     edi, 1
        imul    eax, edx
        cmp     edi, 1
        jne     .L2
.L1:
        ret
```

```
ft(int, int):
        mov     eax, esi
        cmp     edi, 1
        jle     .L14
.L11:
        imul    eax, edi
        sub     edi, 1
        cmp     edi, 1
        jne     .L11
.L14:
        ret
```