Compacting the Uncompactable!

MESH

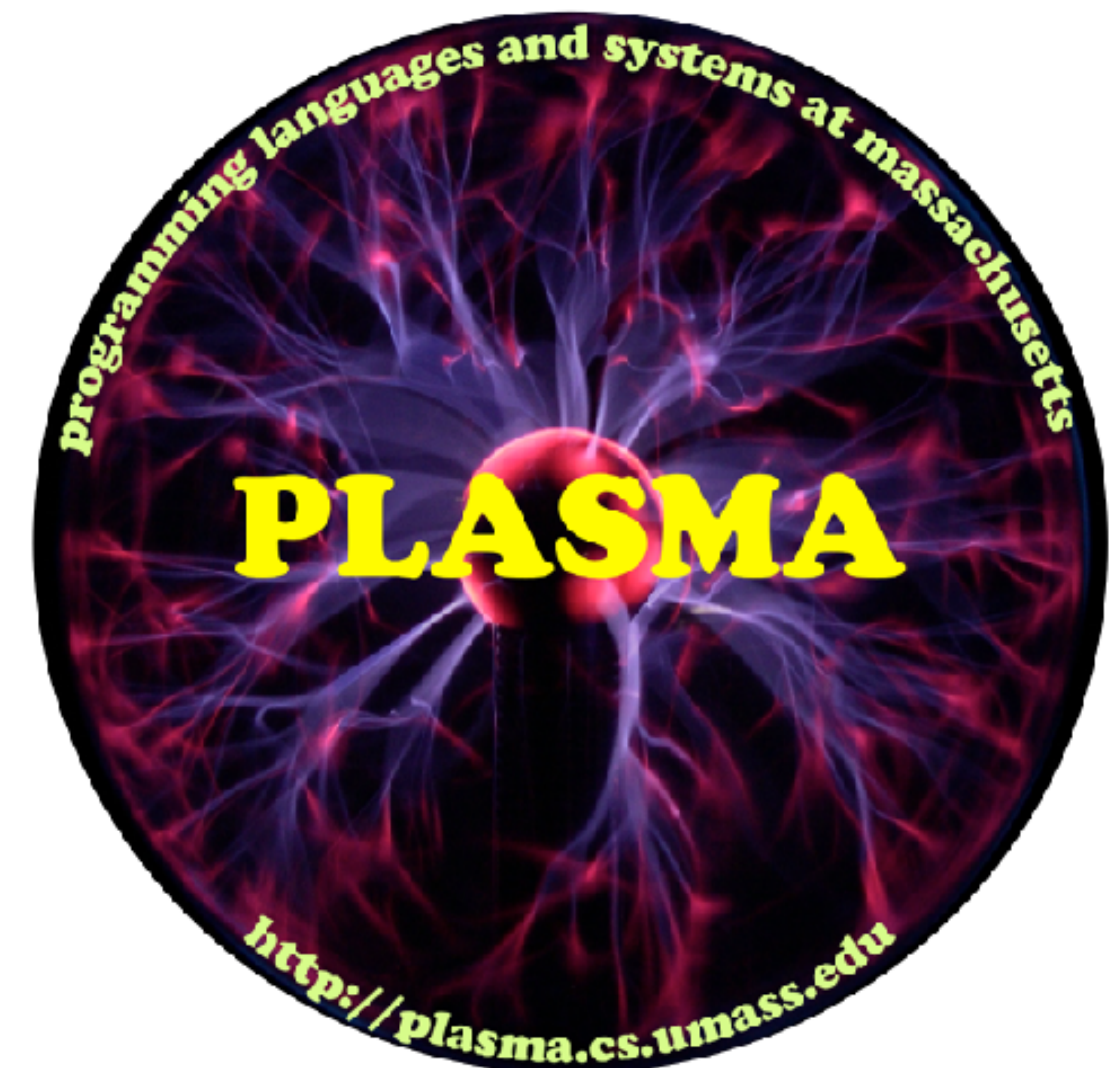
Automatically Compacting Memory for C/C++ Applications

Emery Berger
UMass Amherst

with Bobby Powers, David Tench, & Andrew McGregor
University of Massachusetts Amherst

http://libmesh.org

[PLDI 2019]



PIANO • VOCAL • GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

PIANO · VOCAL · GUITAR

The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

Save
all of Spain!

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

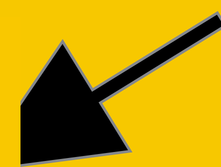
Save
all of Spain!

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

"Reallocate"



MALLOC

~~Save~~ Allocate
all of Spain!

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

(malloc)

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

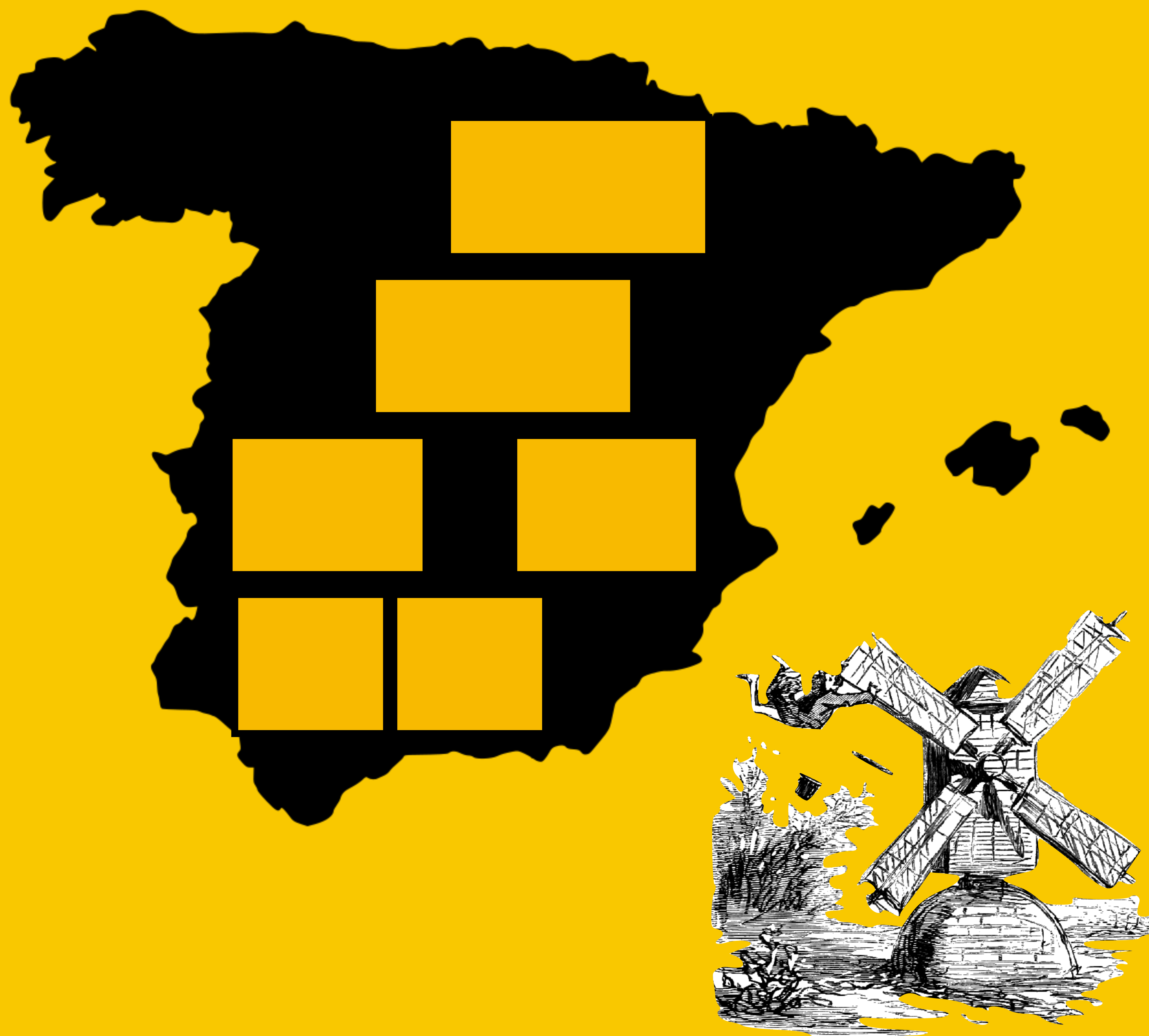
MALLOC

PIANO • VOCAL • GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC



PIANO · VOCAL · GUITAR

The Impossible Dream (The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

?

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

?

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

!

PIANO · VOCAL · GUITAR

The Impossible Dream
(The Quest)

Lyrics by JOE DARIEN
Music by MITCH LEIGH

MALLOC

¡ !

¡ !

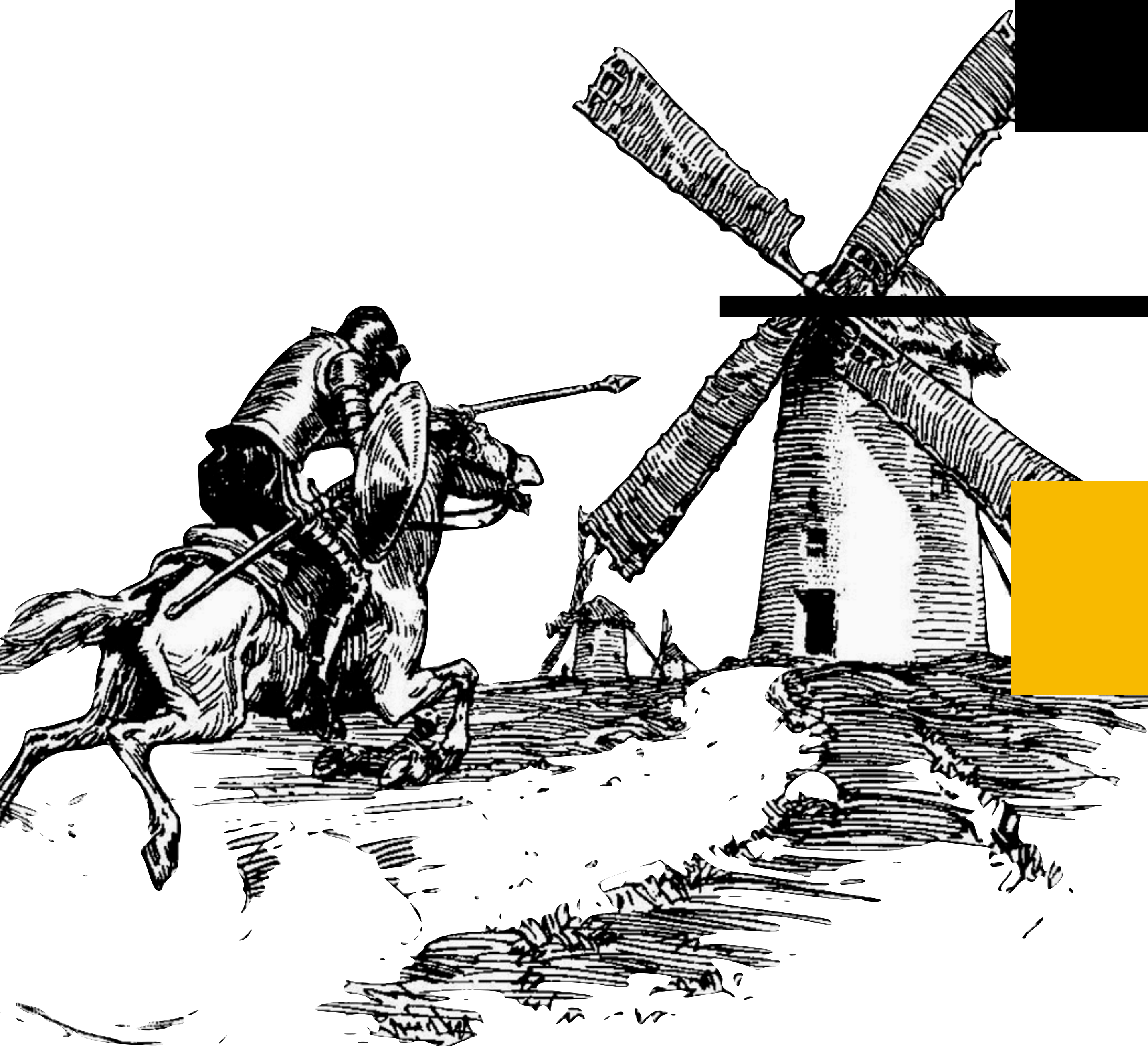
¡Fragmentación!

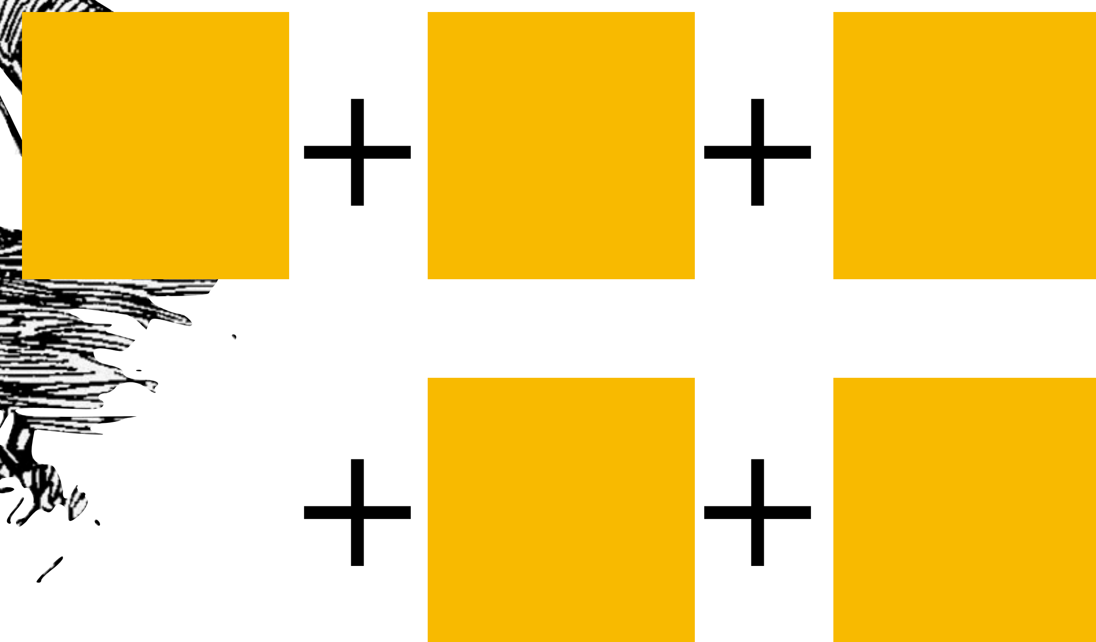
OS Page (“Spain”)

Live objects (malloc’d)

¡Fragmentación!

OS Pages

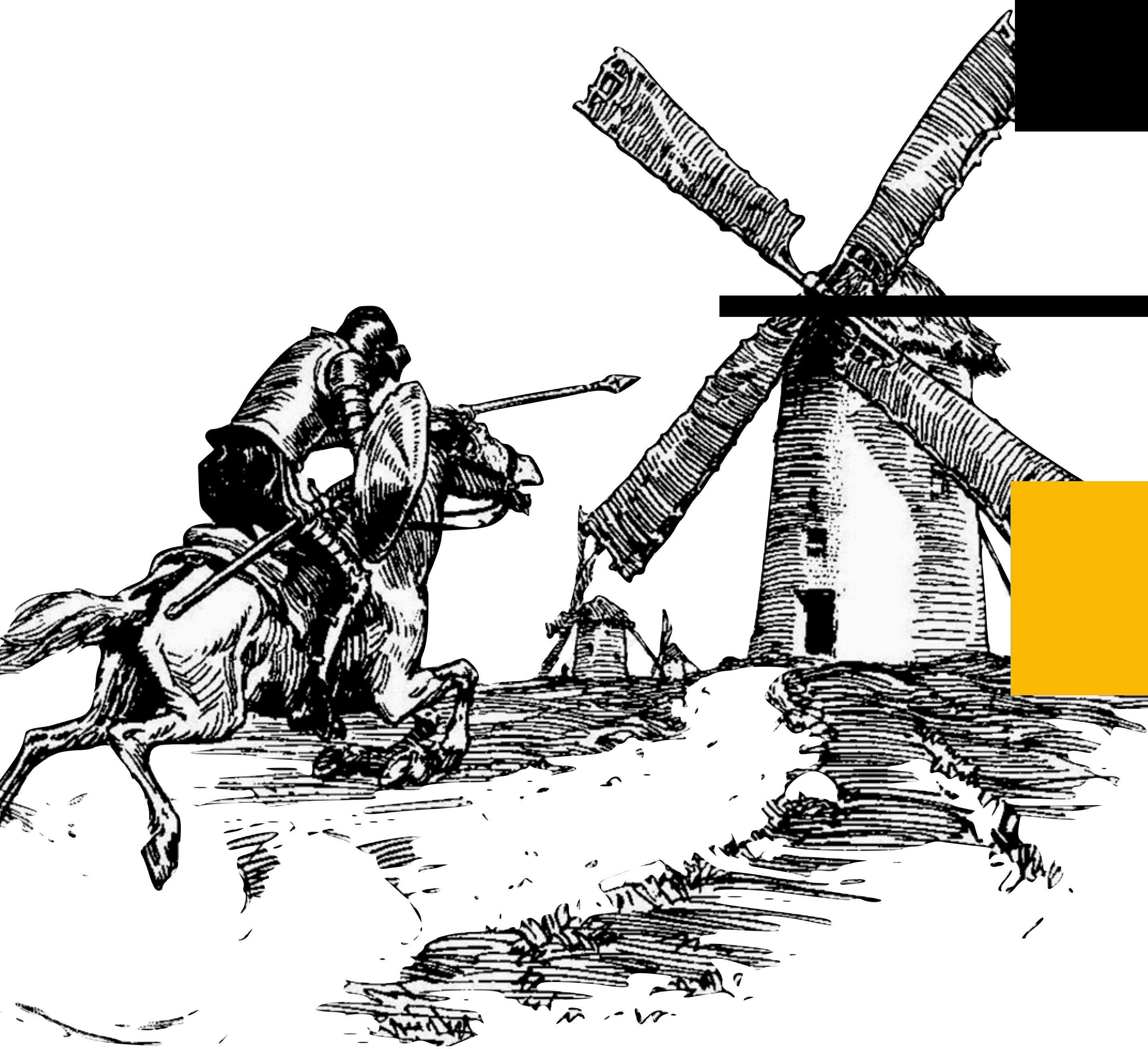




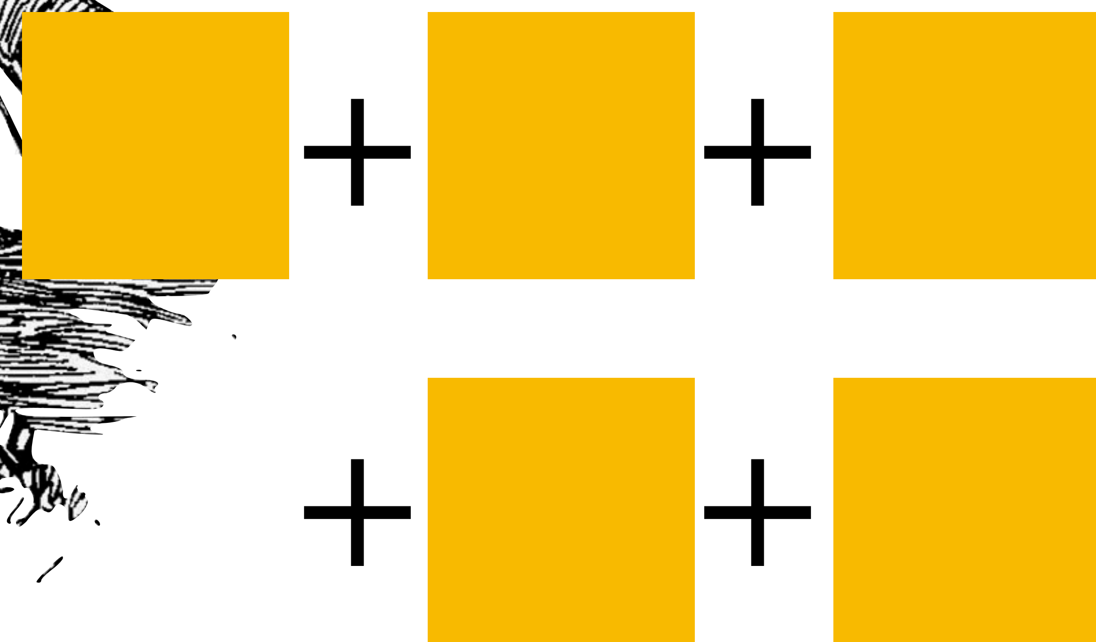
Live objects

¡Fragmentación!

OS Pages



$$\frac{\text{OS Pages}}{\text{Live objects}} \gg 1$$

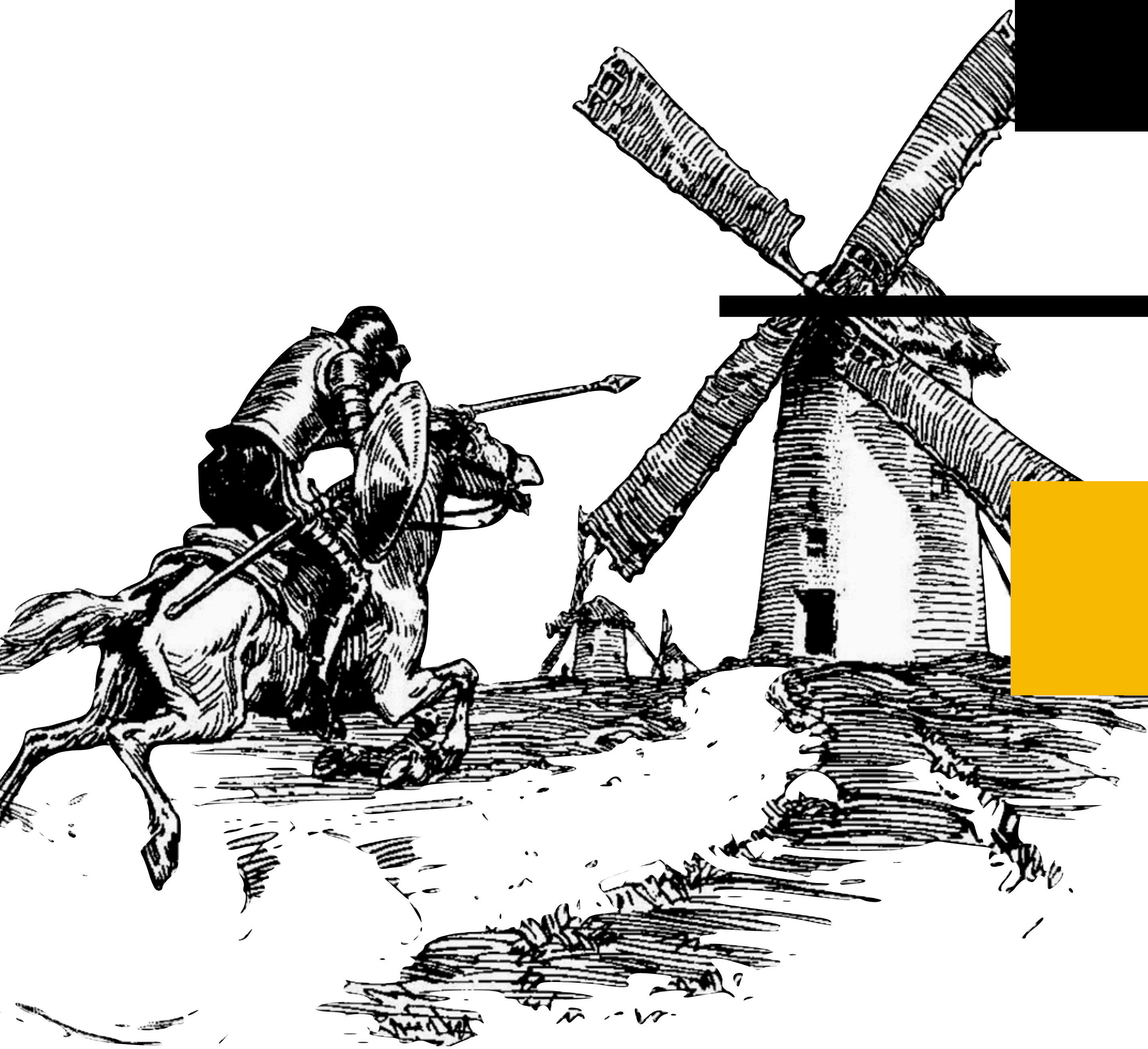


+ + + +

Live objects

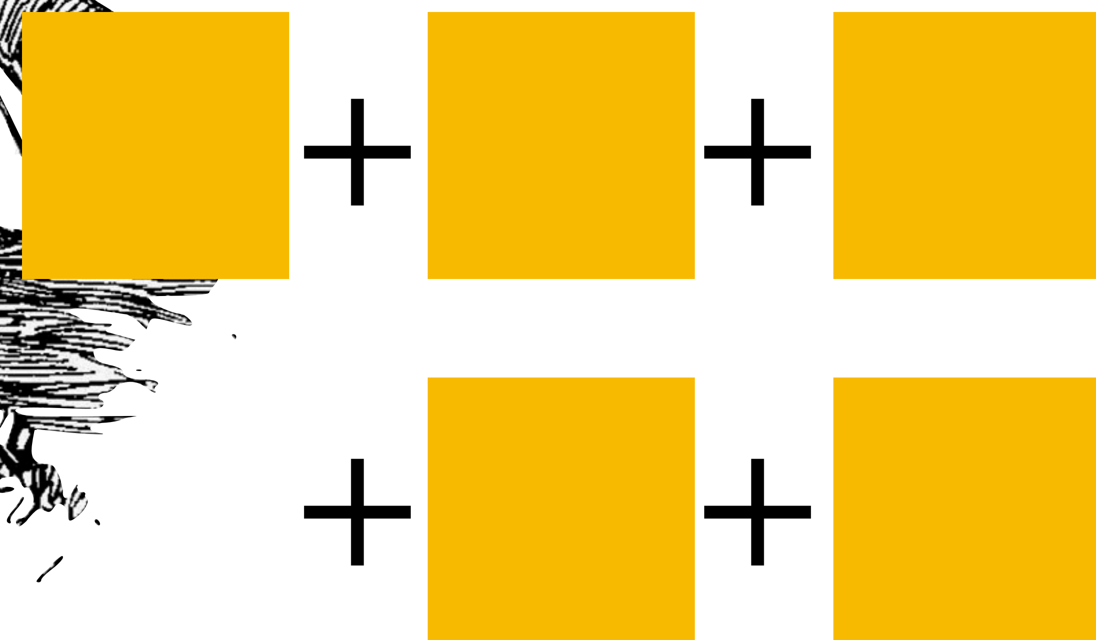
¡Fragmentación!

OS Pages



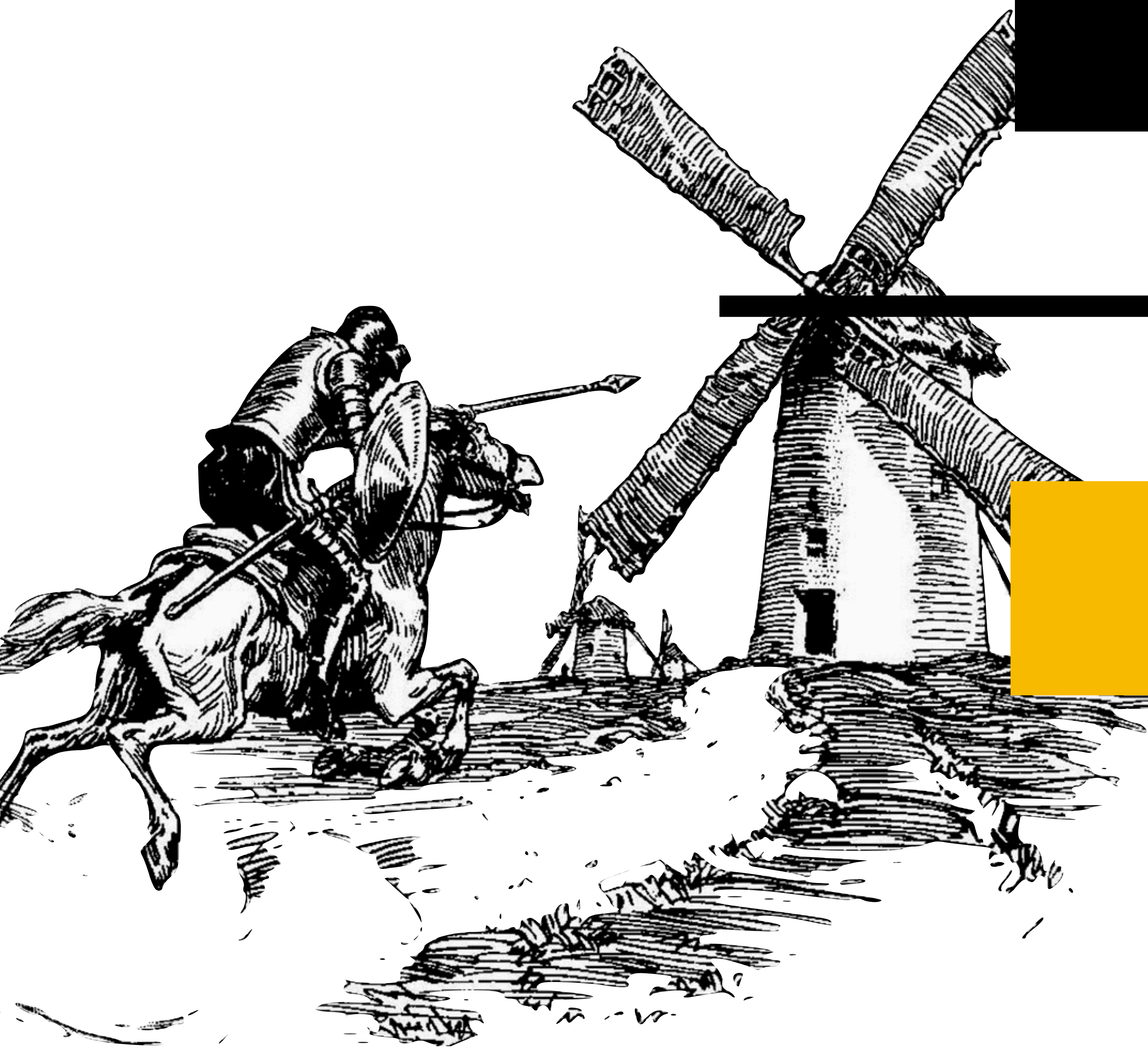
O(log OS Pages / Live objects) 13x!

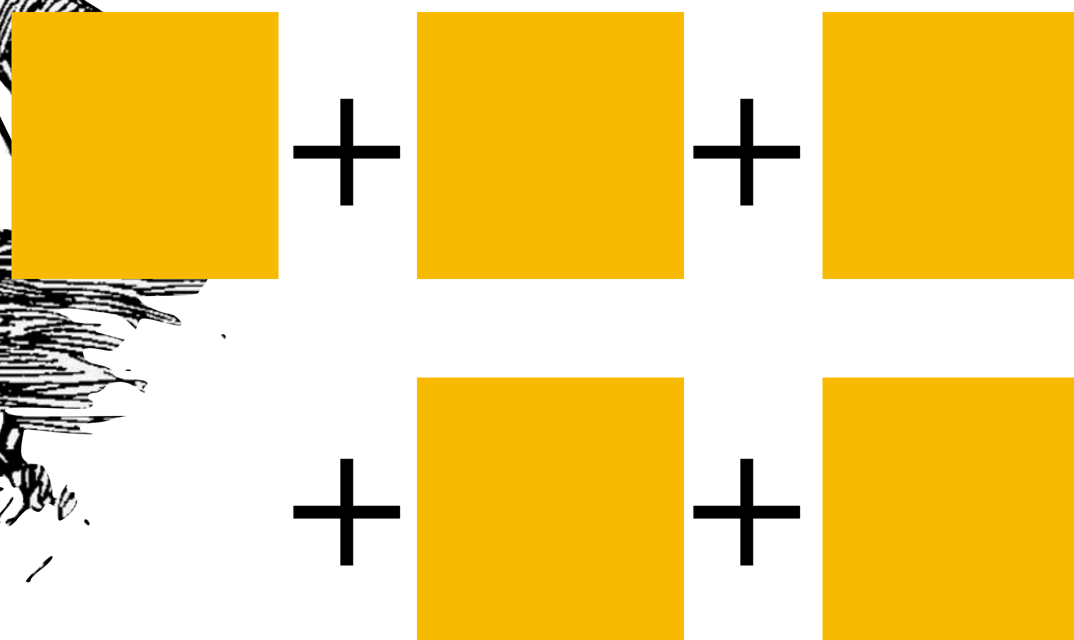
[Robson '77]



Live objects

¿Compactación?





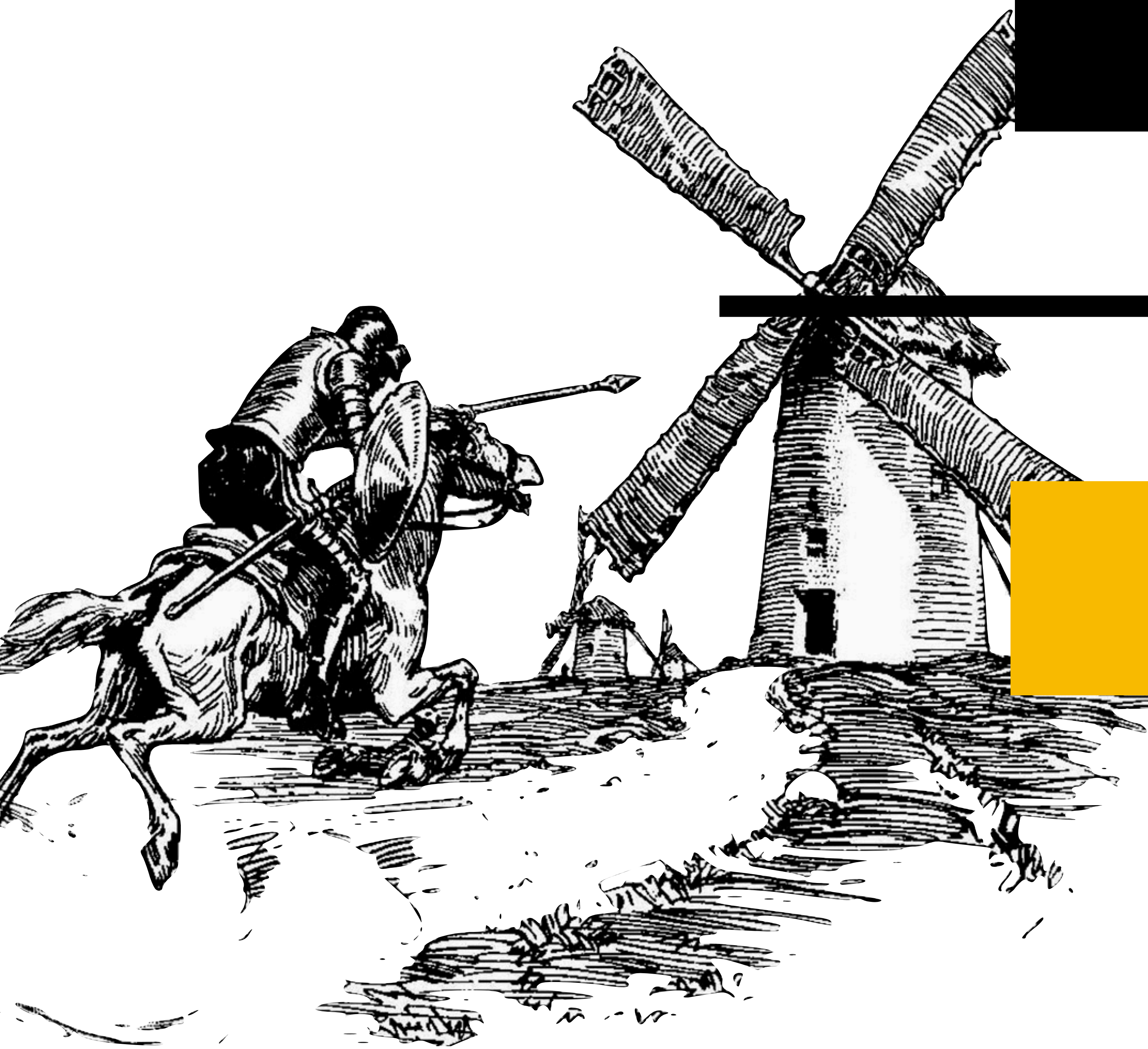
OS Pages

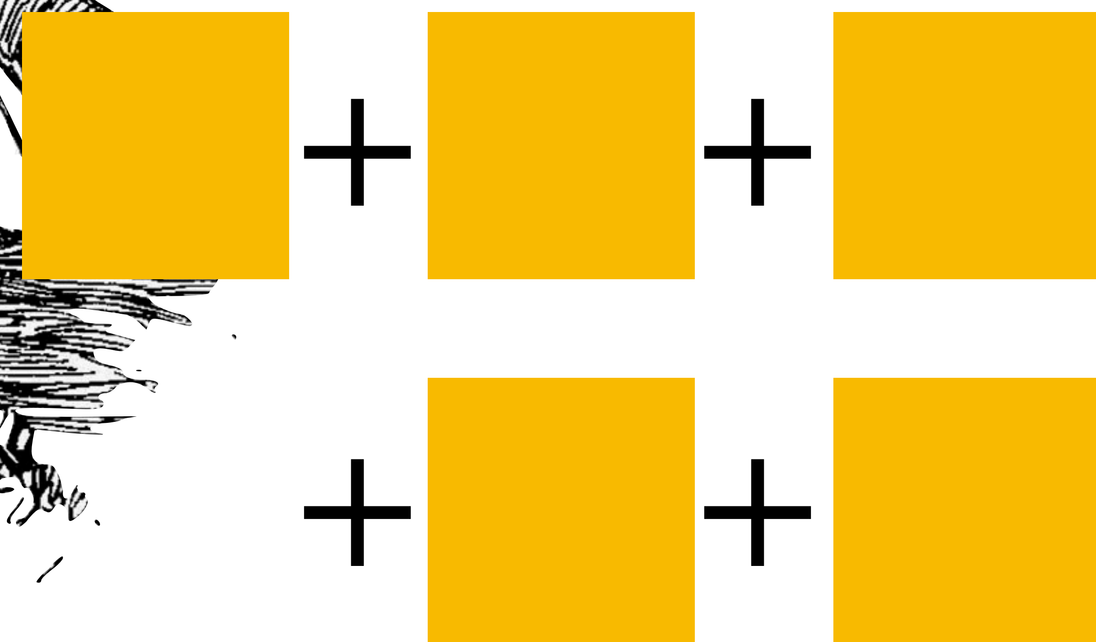
O(log) 13x!

[Robson ’77]

Live objects

¿Compactación?

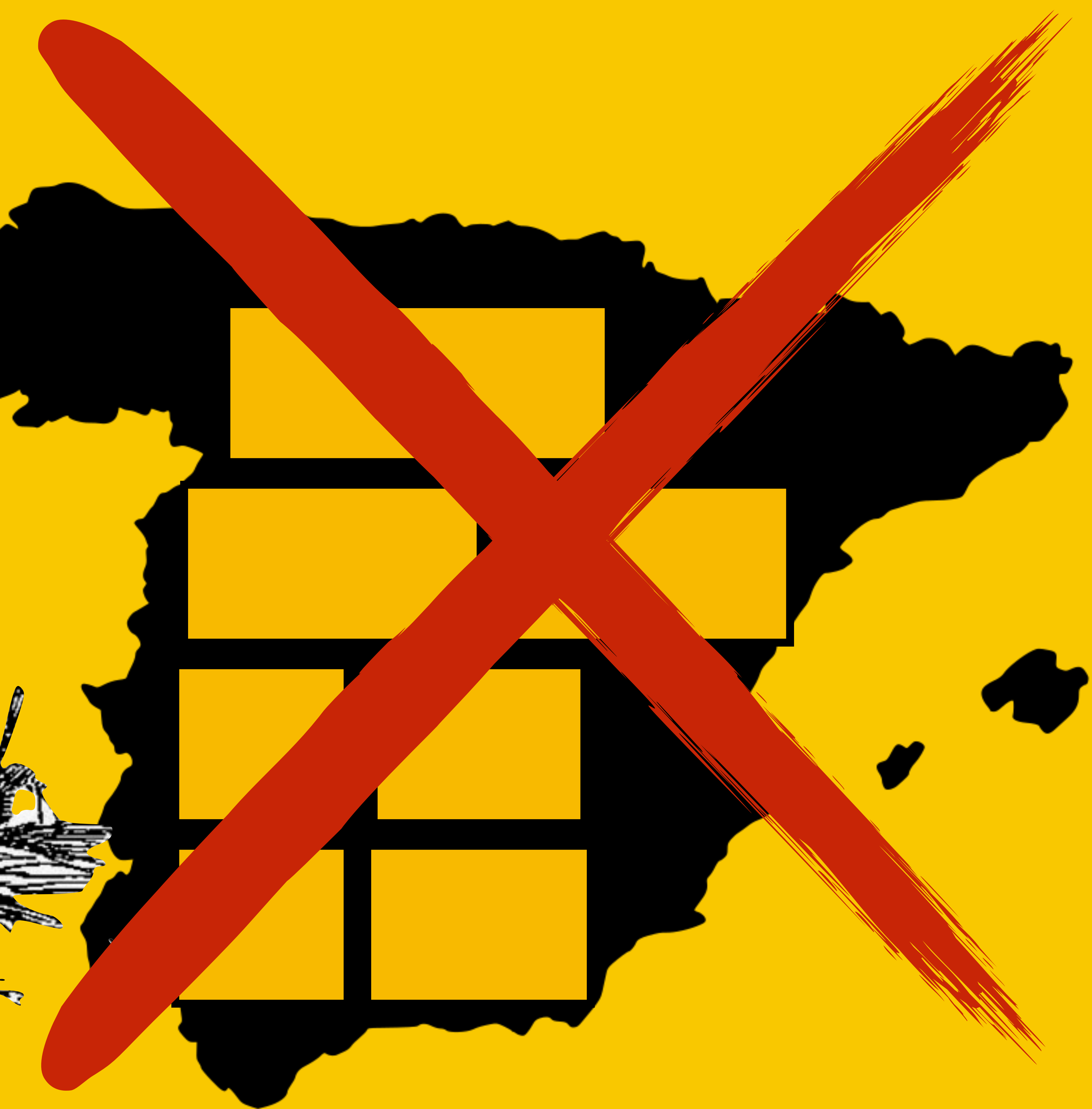




OS Pages

O(1)

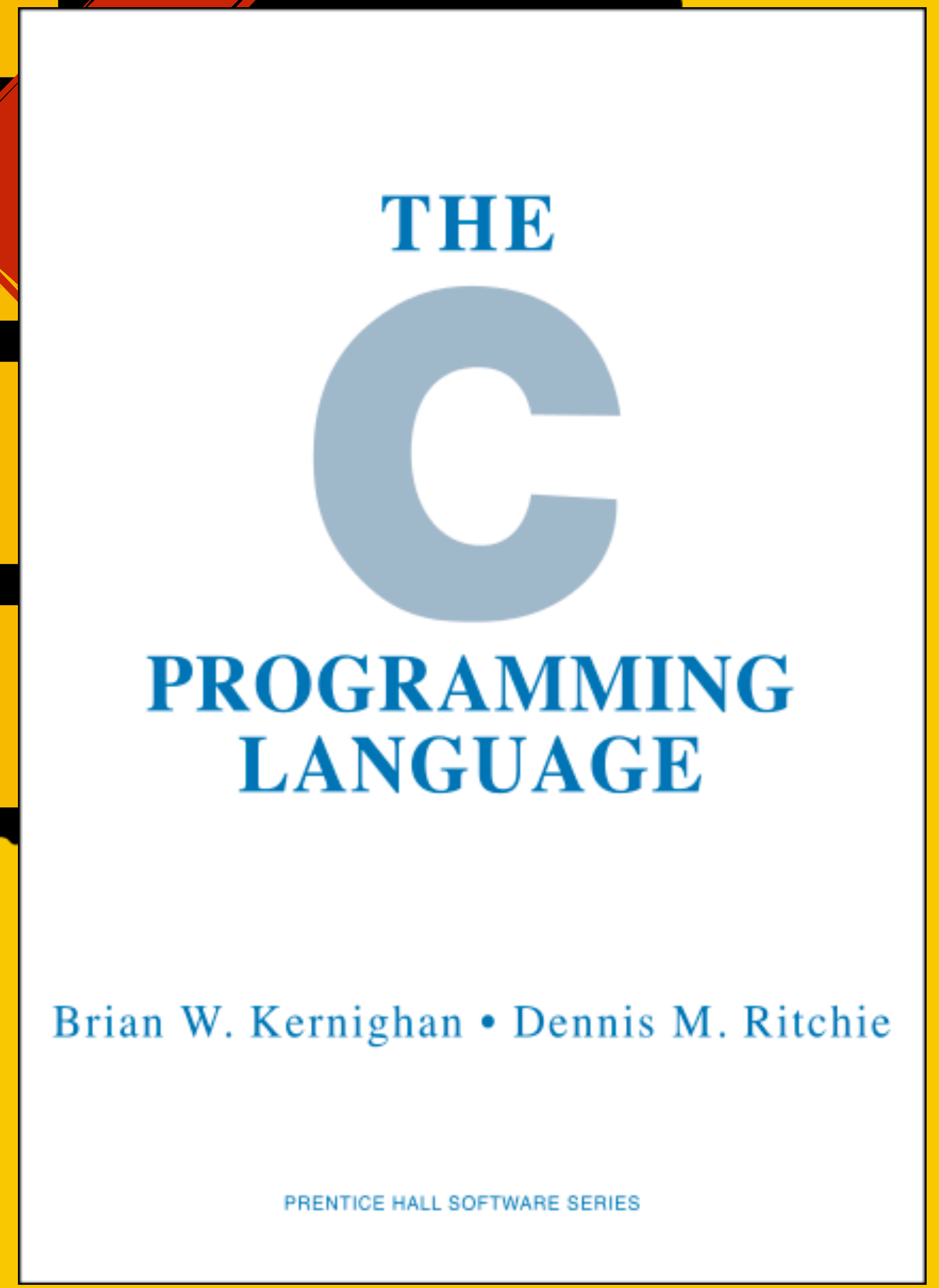
Live objects

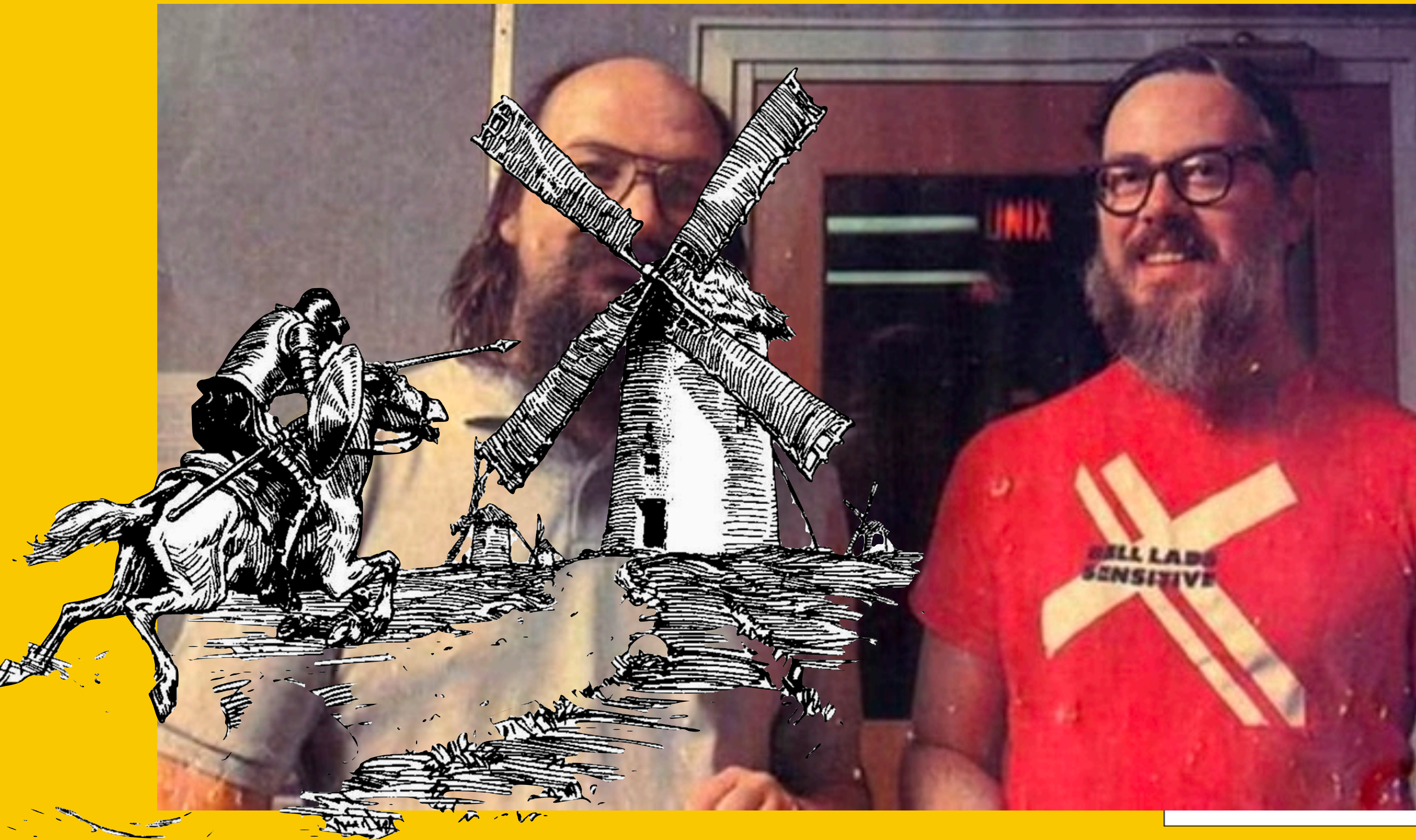


¡ !

UNIX

UNIX
BELL LABS
SENSITIVE


THE
C
PROGRAMMING
LANGUAGE
Brian W. Kernighan • Dennis M. Ritchie
PRENTICE HALL SOFTWARE SERIES


M. Ritchie

NG
M. Ritchie

THE C PROGRAMMING LANGUAGE

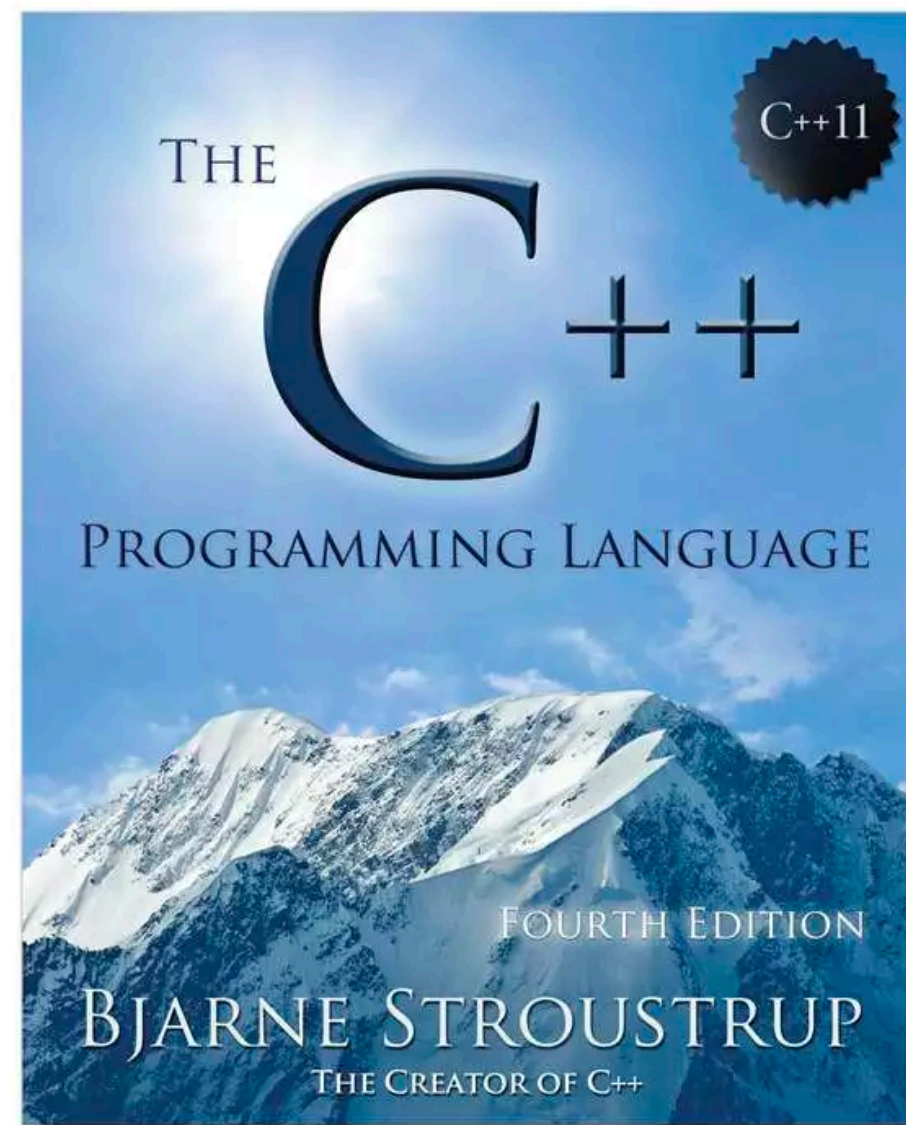
Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES

THE
C
PROGRAMMING
LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES



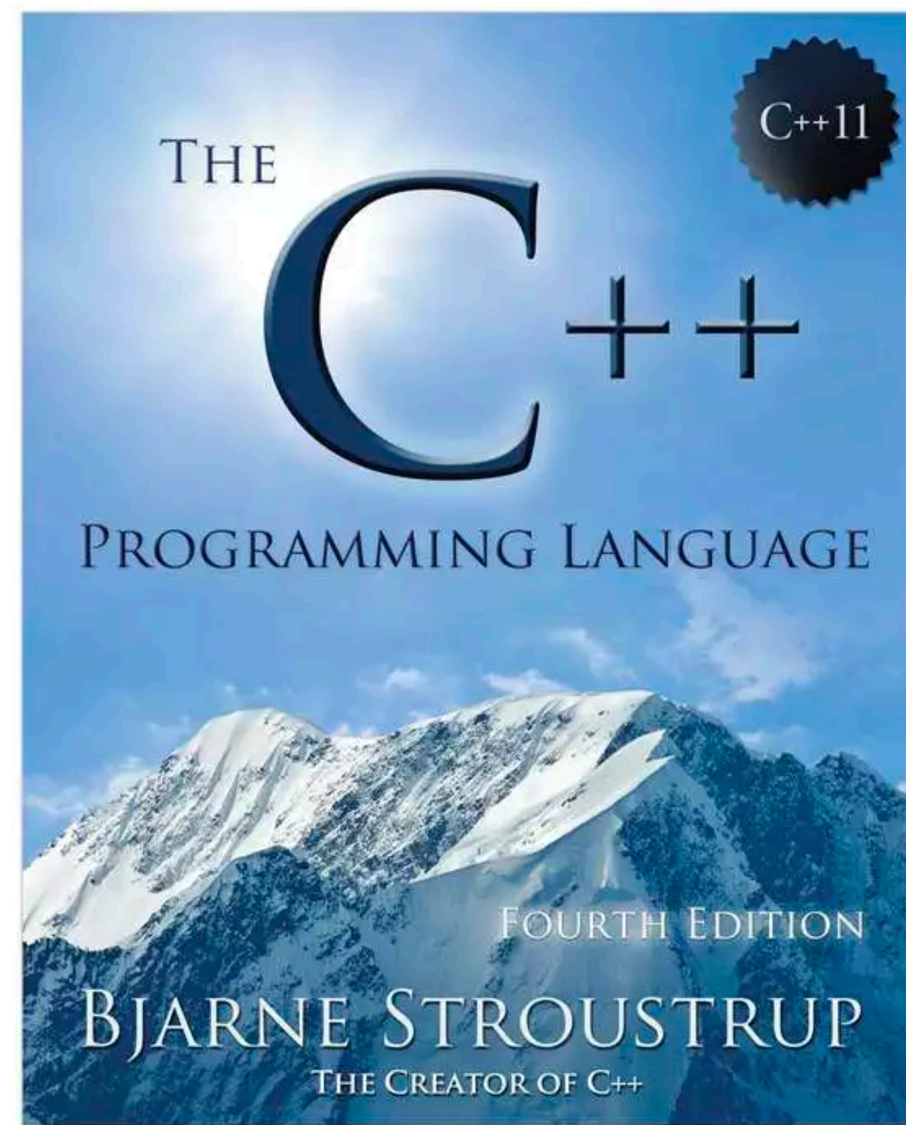
THE

C

PROGRAMMING
LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

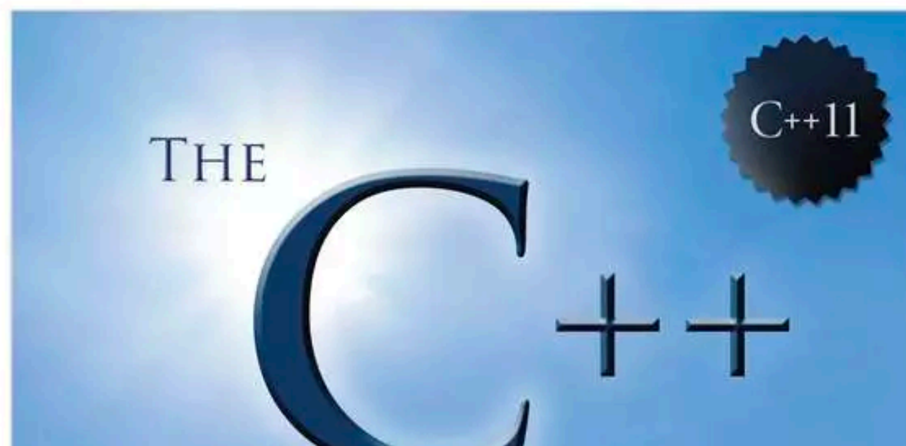
PRENTICE HALL SOFTWARE SERIES



THE C PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES



[Lattner, 2016]

Why not a tracing GC?

- Native interoperability with unmanaged code
- Deterministic destruction provides:
  - No "finalizer problems" like resurrection, threading, etc.
  - Deterministic performance: can test/debug performance stutters
- Performance:
  - GC use ~3-4x more memory than ARC to achieve good performance
  - Memory usage is very important for mobile and cloud apps
  - Incremental/concurrent GCs slow the mutator like ARC does

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management
Matthew Hertz, Emery D. Berger. OOPSLA'05

- Native interoperability with unmanaged code
- Deterministic destruction provides:
  - No "finalizer problems" like resurrection, threading, etc.
  - Deterministic performance: can test/debug performance stutters
- Performance:
  - GC use ~3-4x more memory than ARC to achieve good performance
  - Memory usage is very important for mobile and cloud apps
  - Incremental/concurrent GCs slow the mutator like ARC does

Quantifying the Performance of Garbage Collection vs. Explicit Memory Management
Matthew Hertz, Emery D. Berger. OOPSLA'05

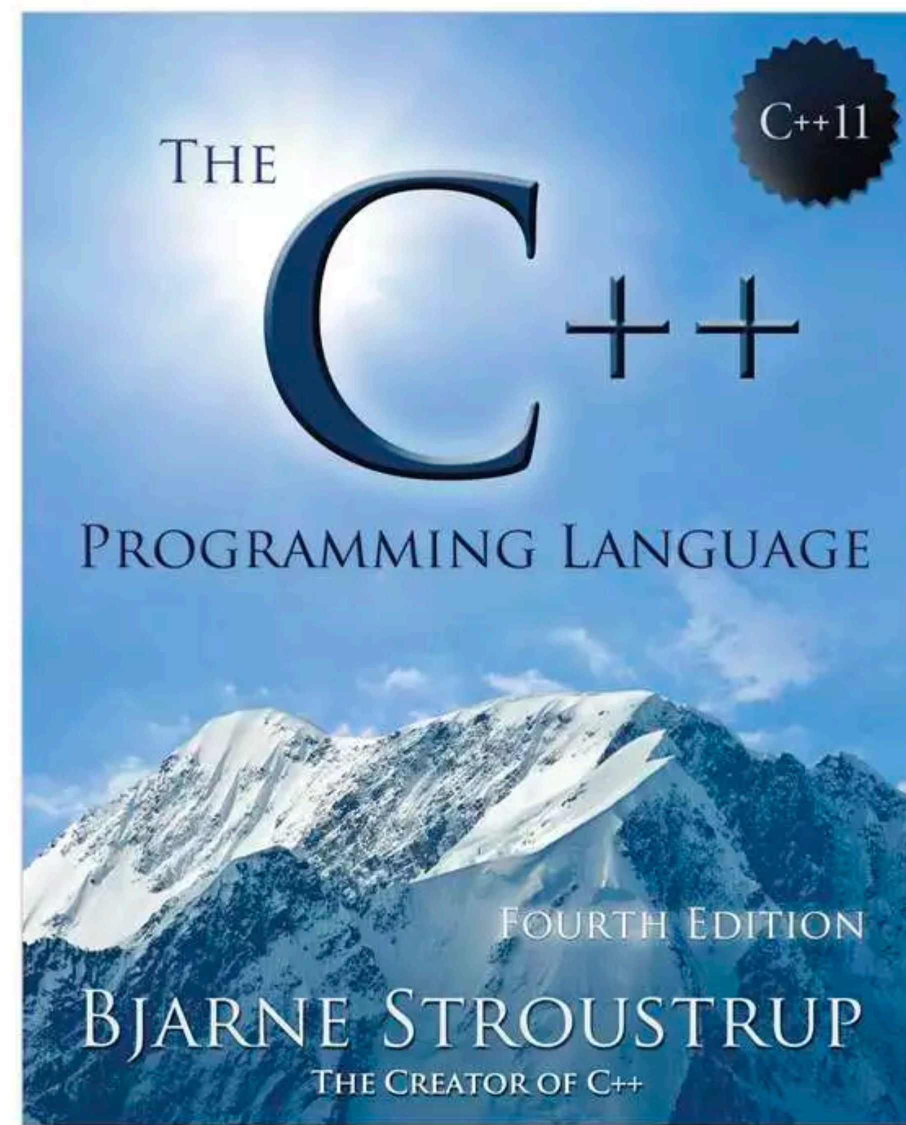
THE

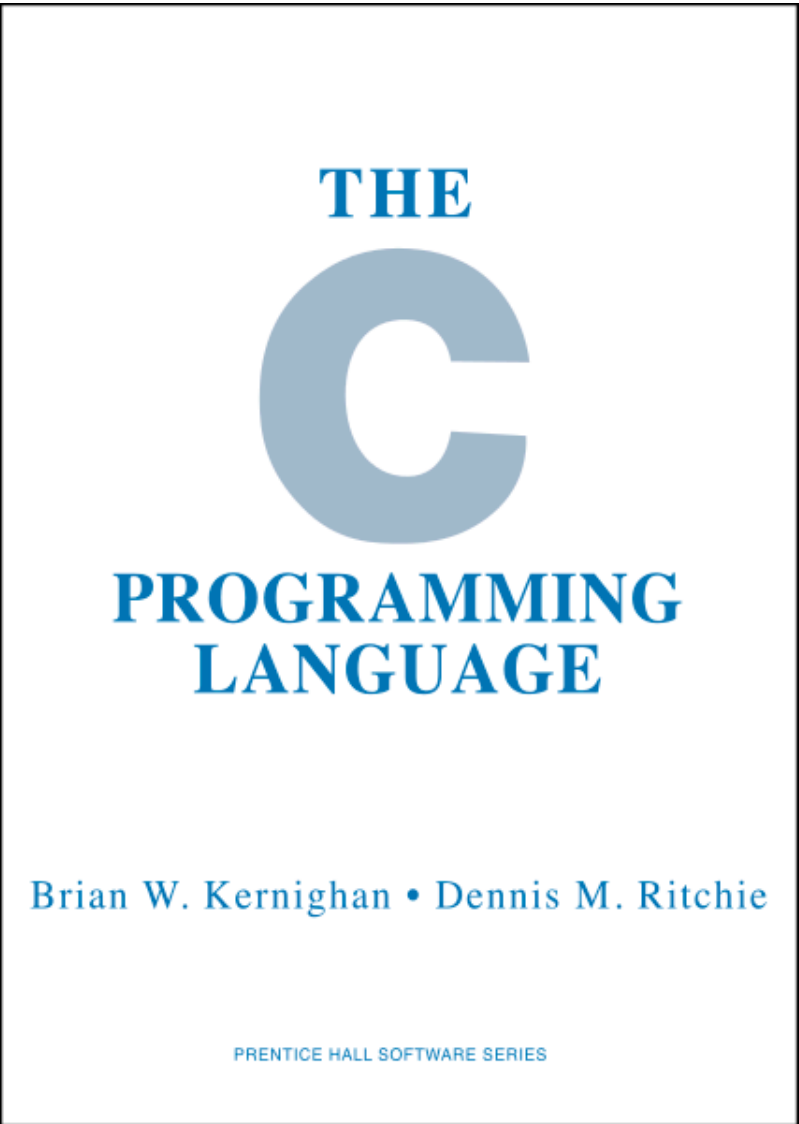
C

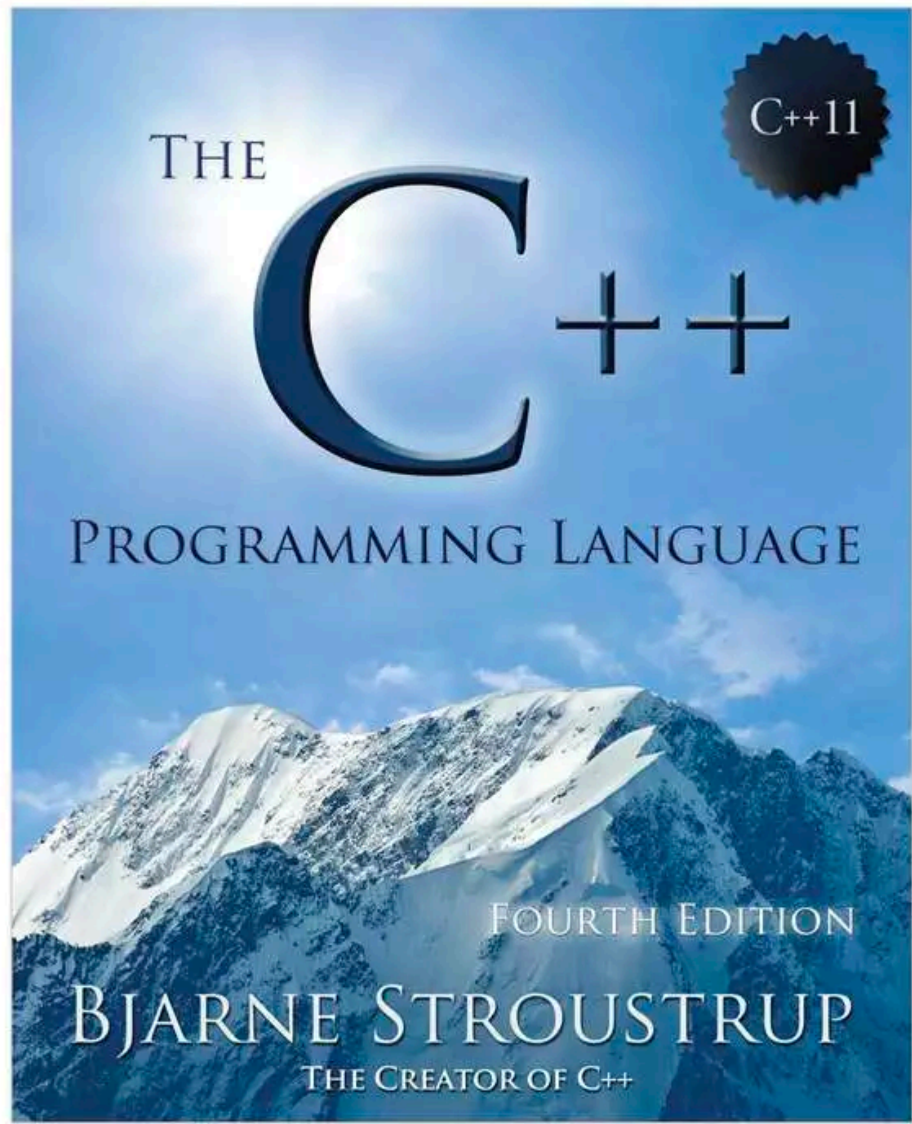
PROGRAMMING
LANGUAGE

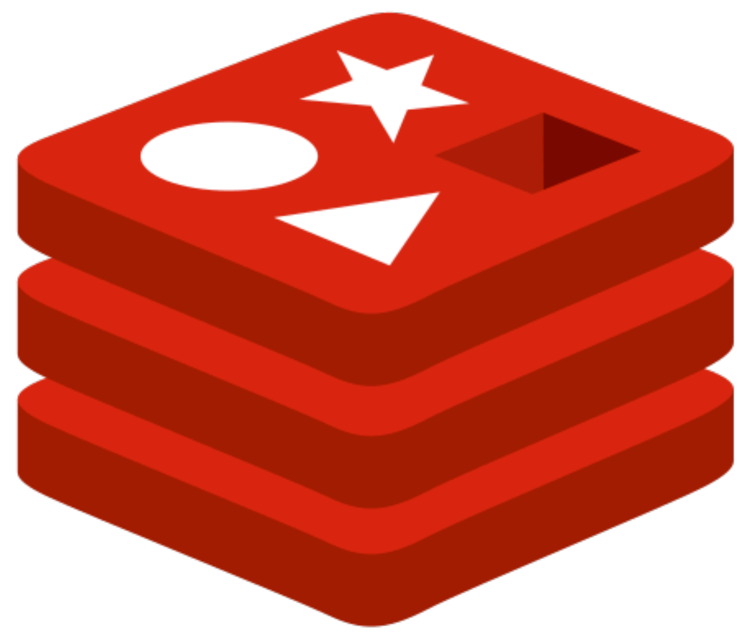
Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES

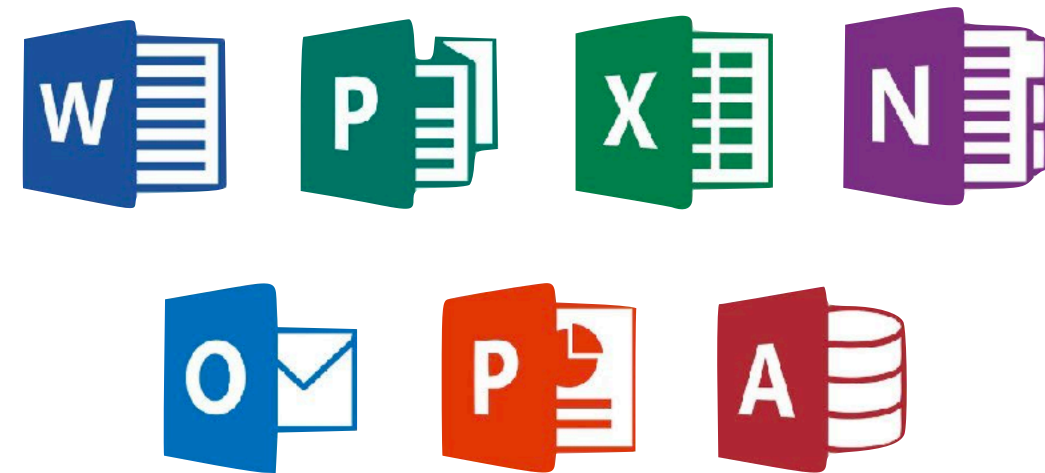


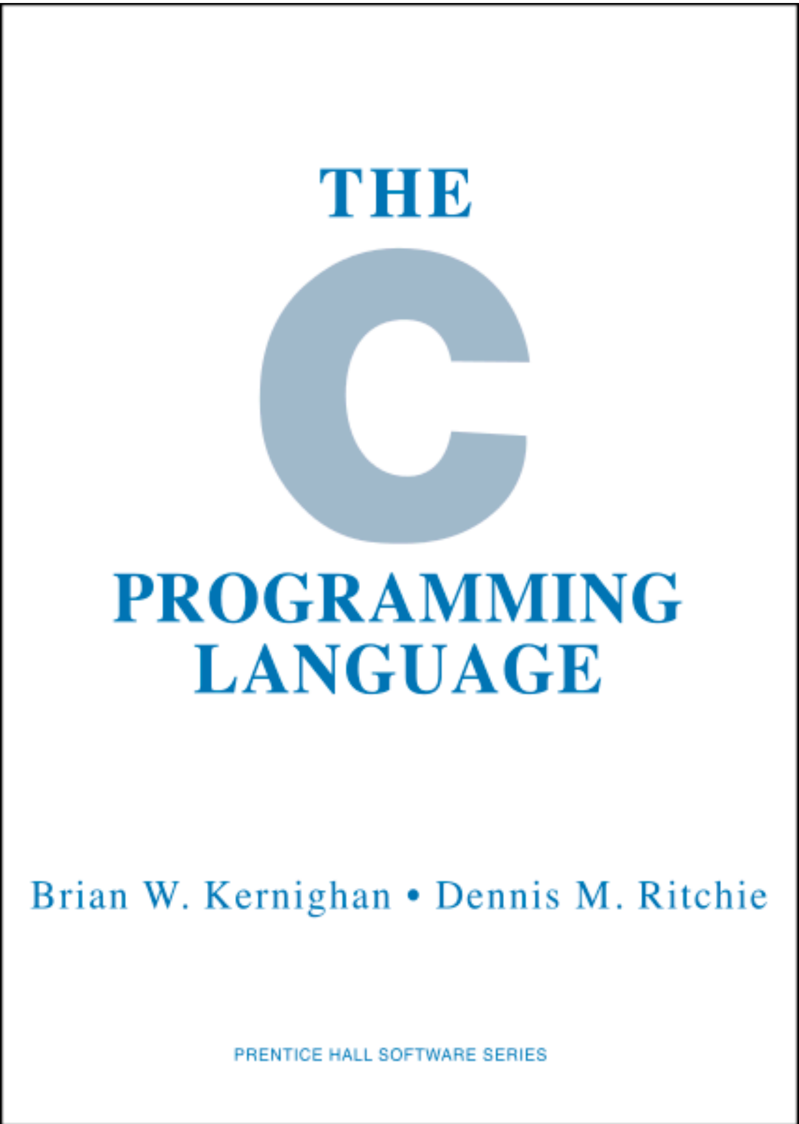
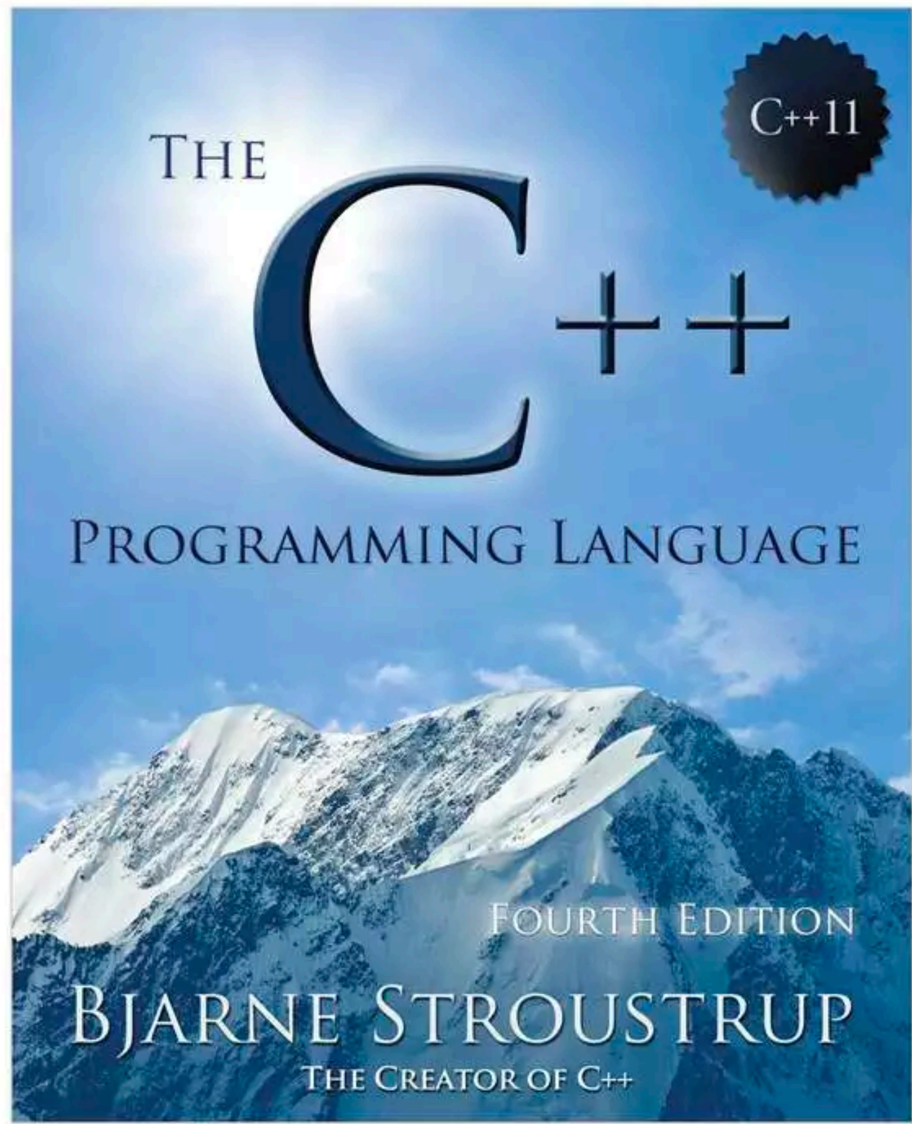


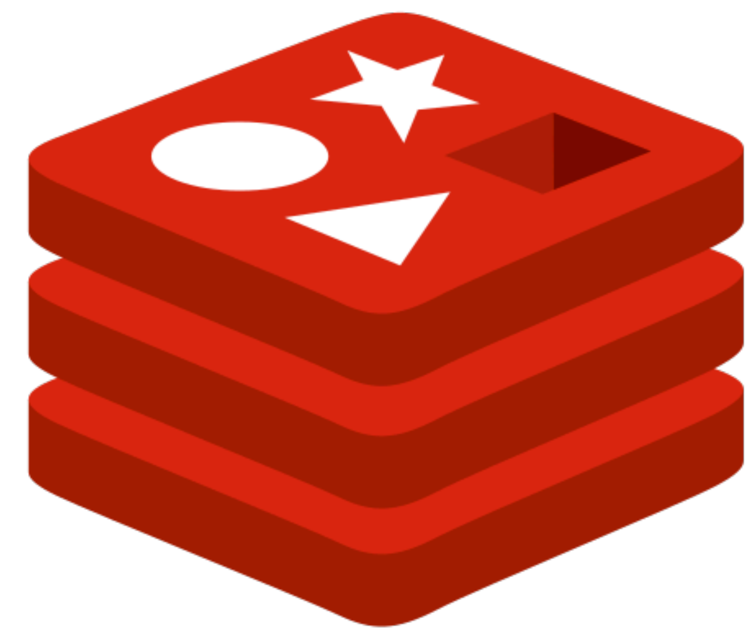


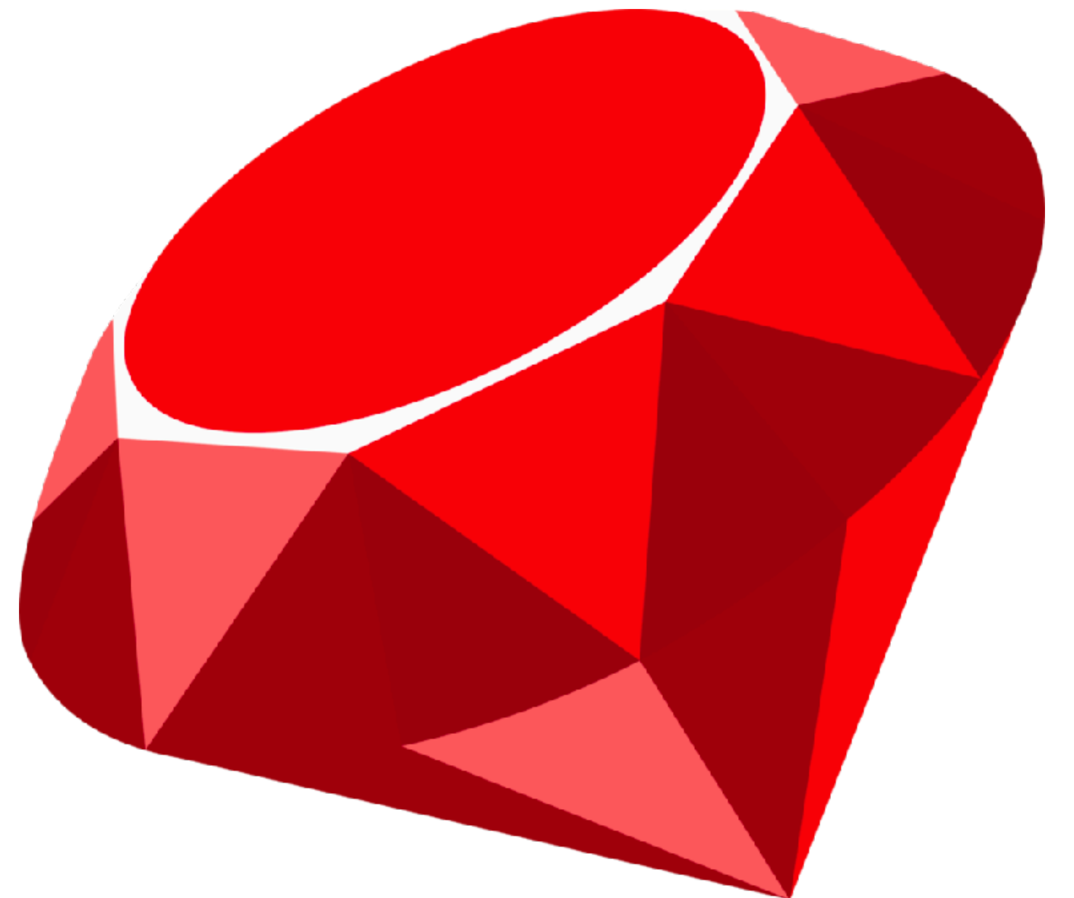


redis




THE
C
PROGRAMMING LANGUAGE
Brian W. Kernighan • Dennis M. Ritchie
PRENTICE HALL SOFTWARE SERIES

C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

The Swift Programming Language
Swift 5 Edition

redis

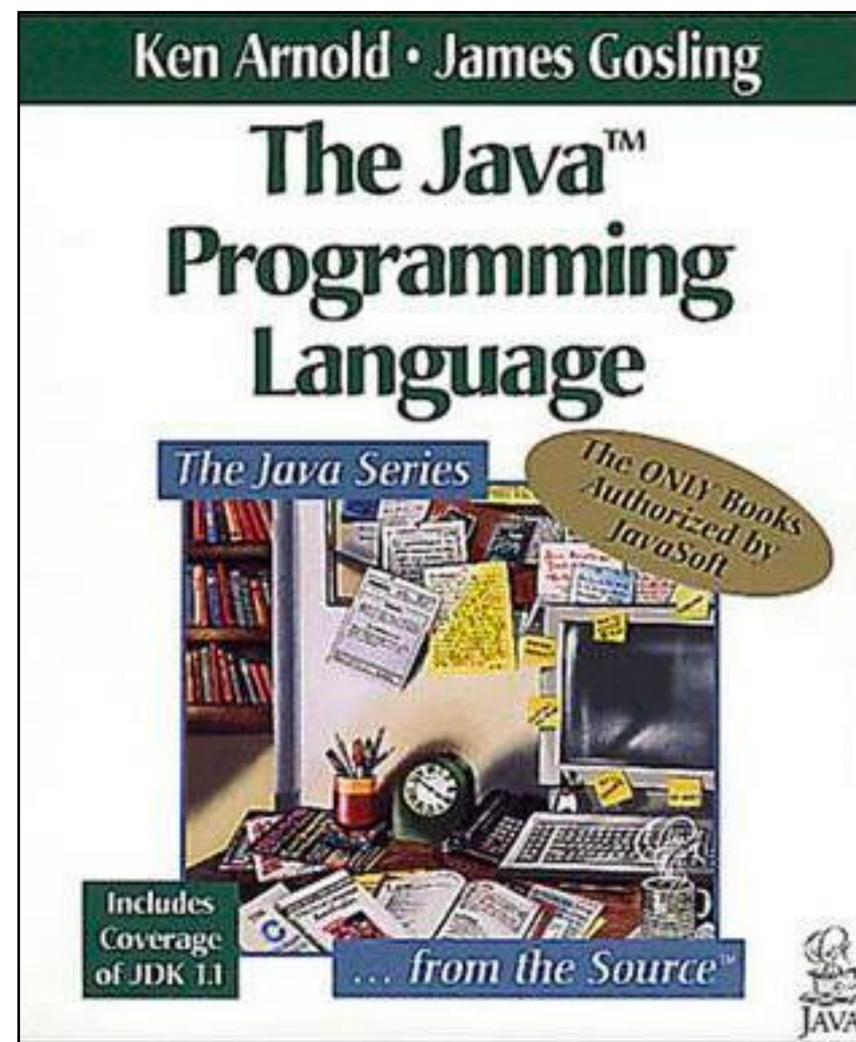


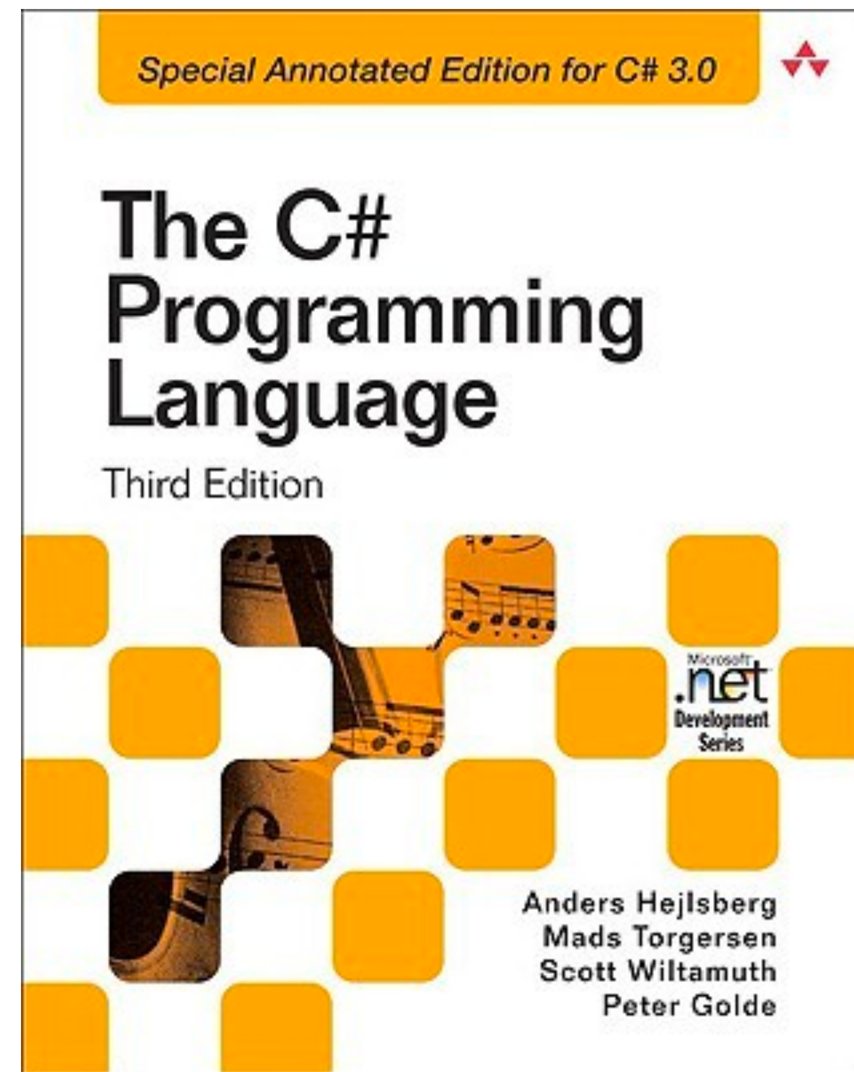
MySQL™

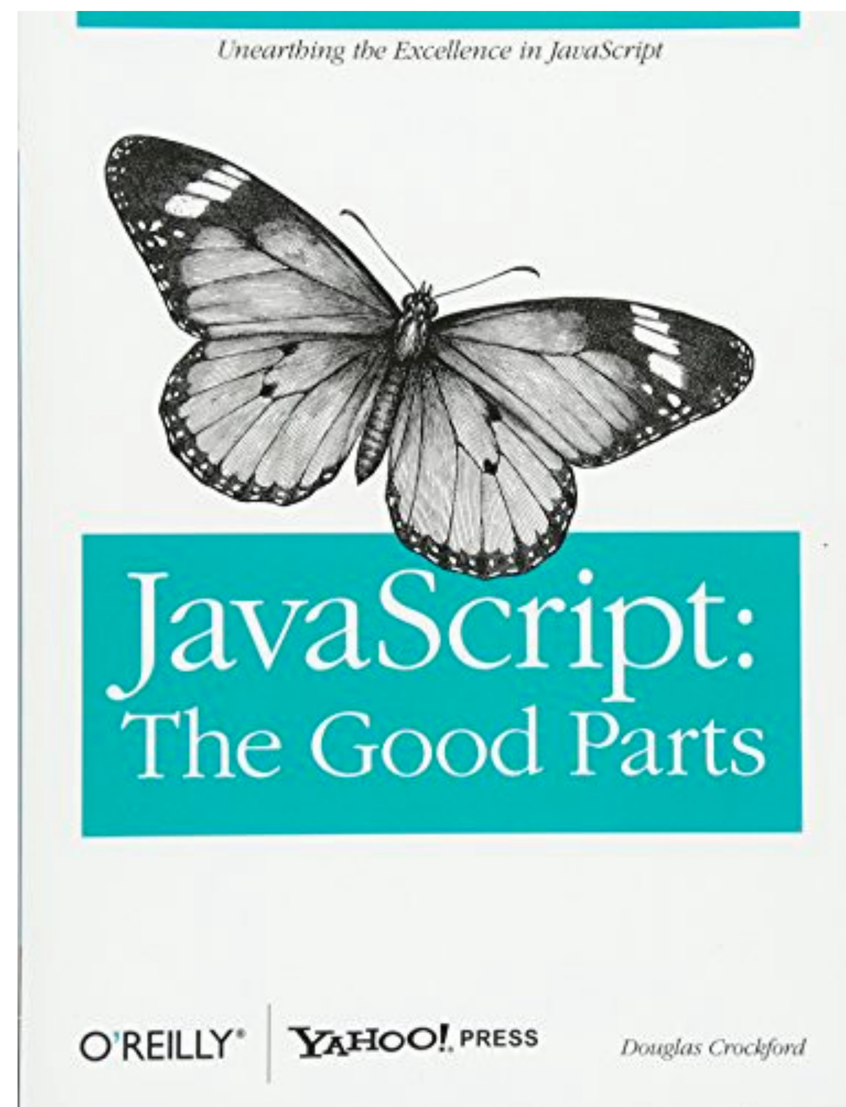
Office

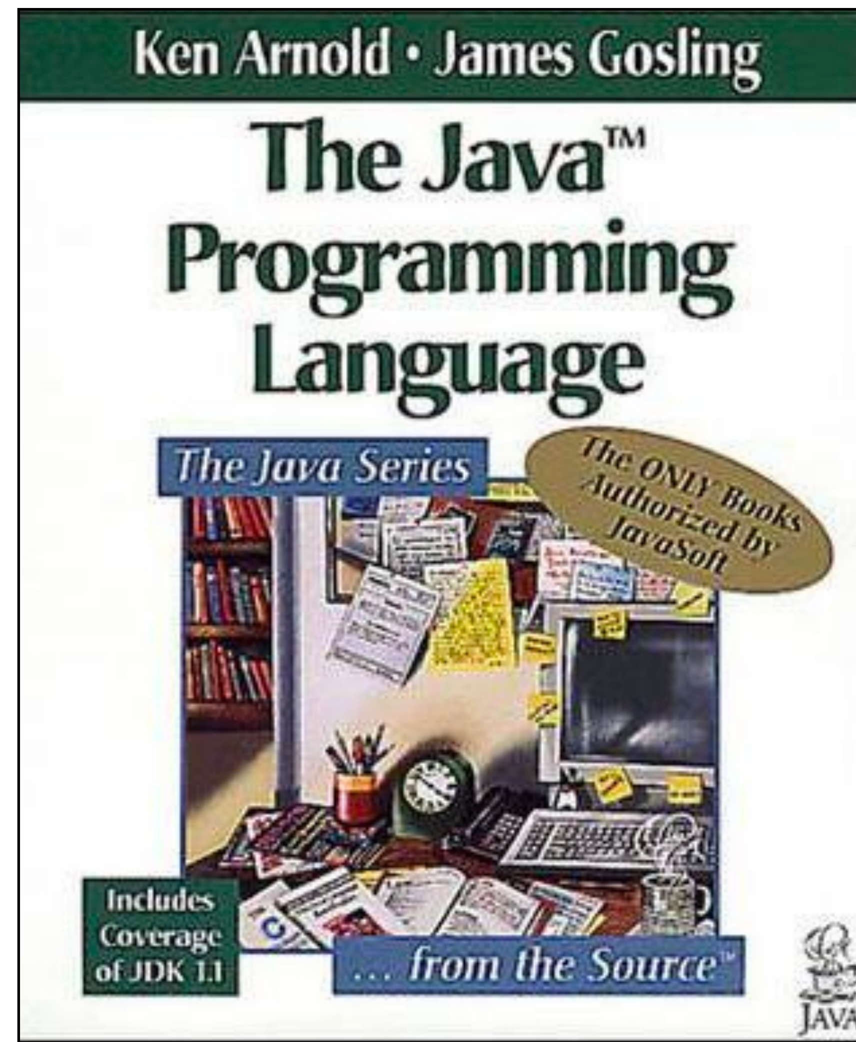
W
P
X
N

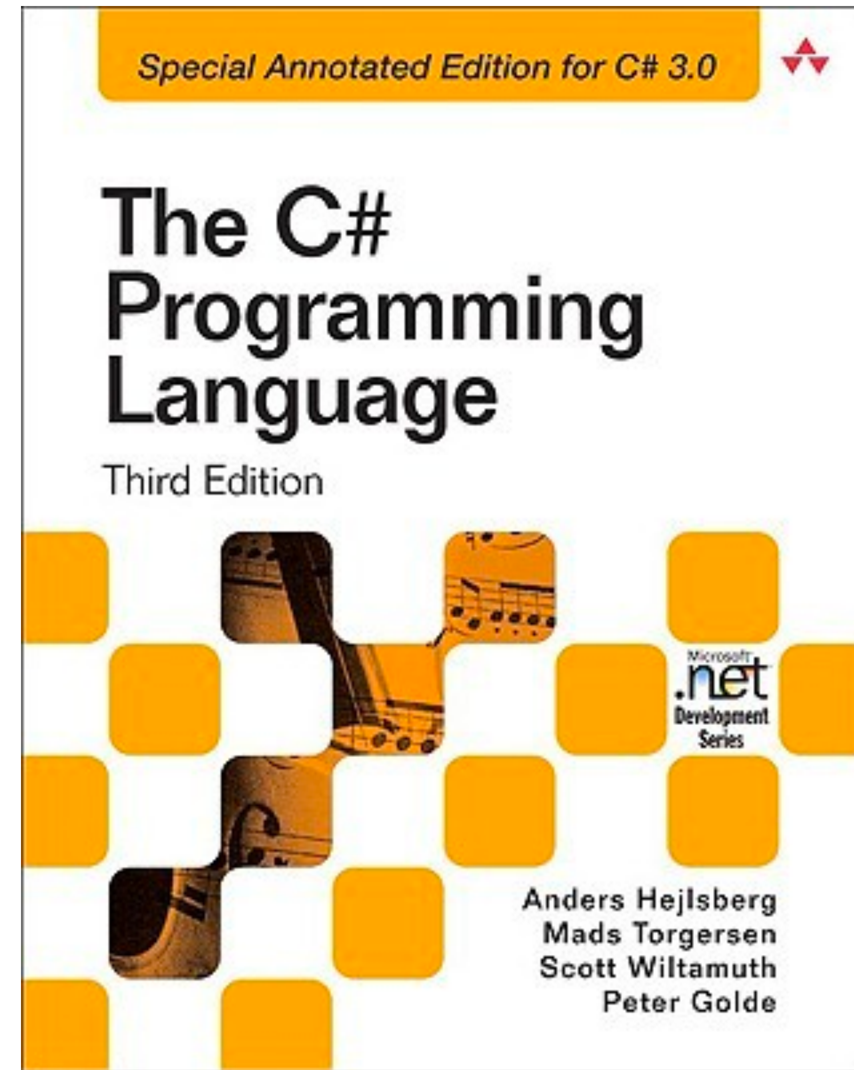
O
P
A

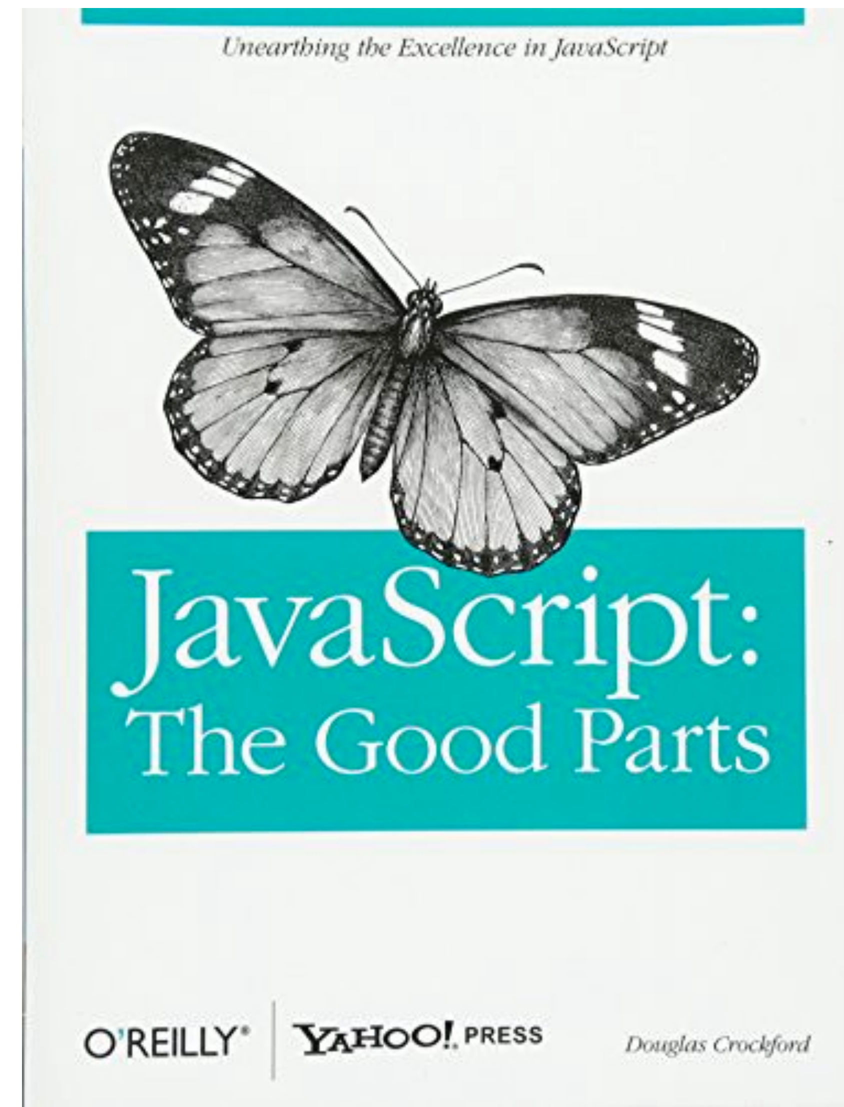

Ken Arnold • James Gosling
The Java™ Programming Language
The Java Series
The ONLY Books Authorized by JavaSoft
Includes Coverage of JDK 1.1
... from the Source™
JAVA


Special Annotated Edition for C# 3.0
The C# Programming Language
Third Edition
.net Development Series
Anders Hejlsberg
Mads Torgersen
Scott Wiltamuth
Peter Golde


Unearthing the Excellence in JavaScript
JavaScript: The Good Parts
O'REILLY®
YAHOO! PRESS
Douglas Crockford

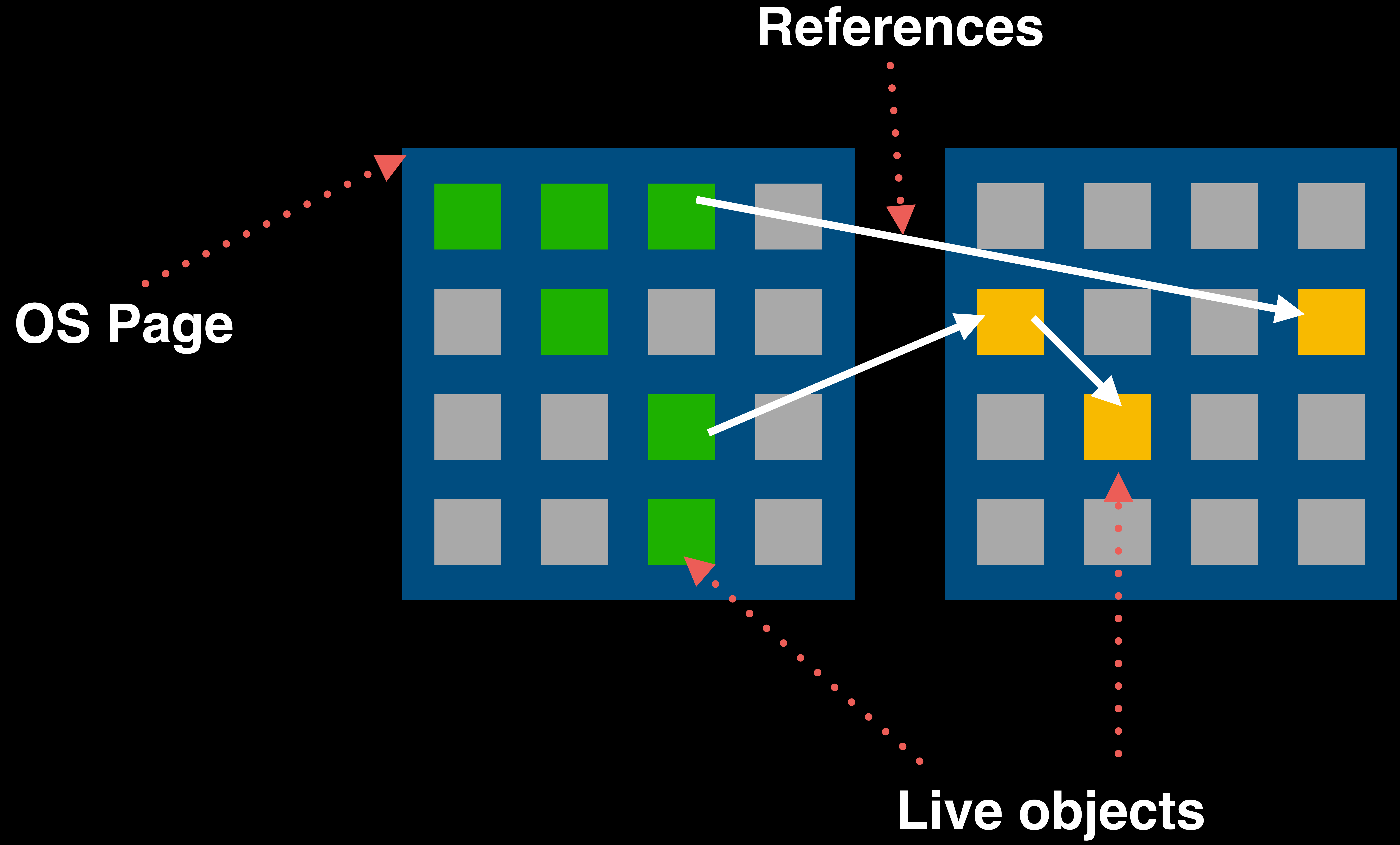


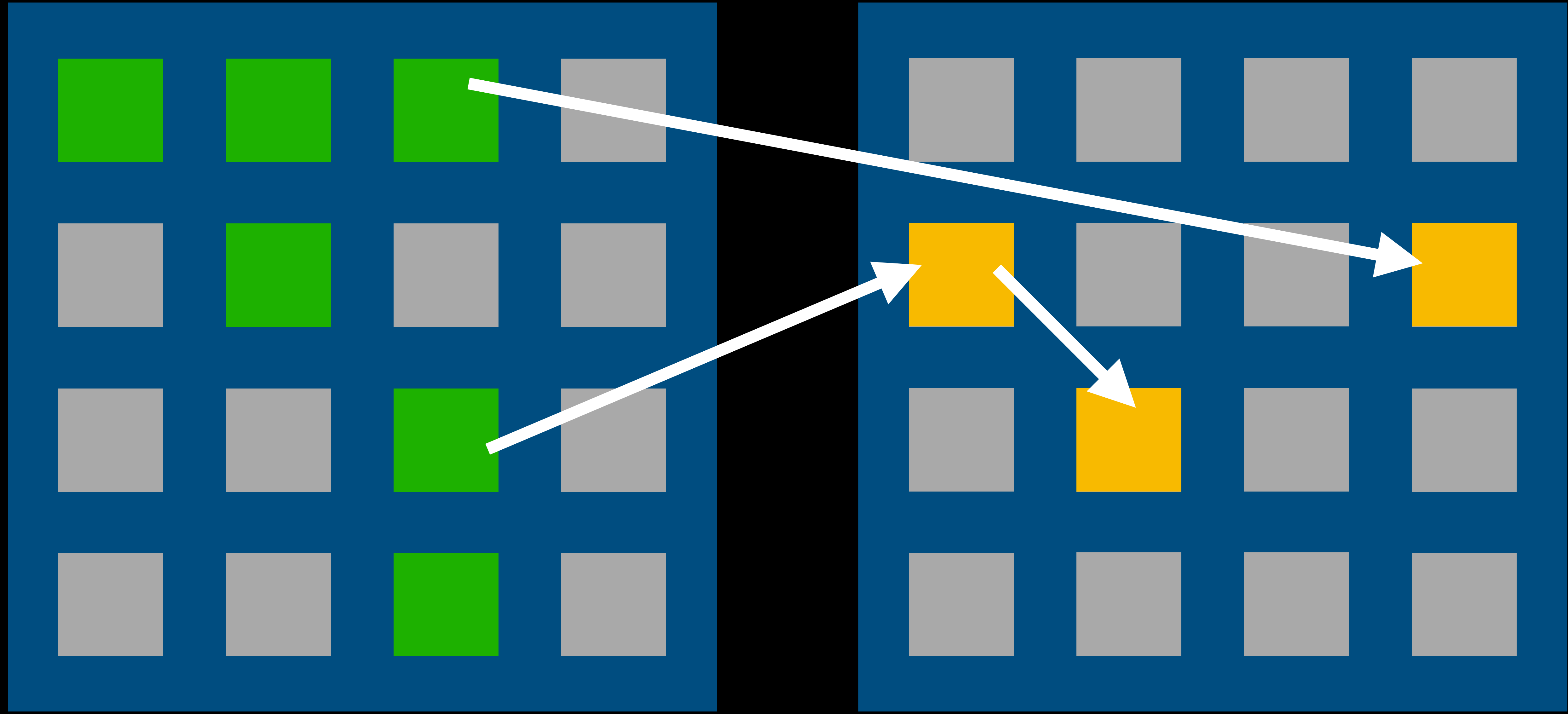


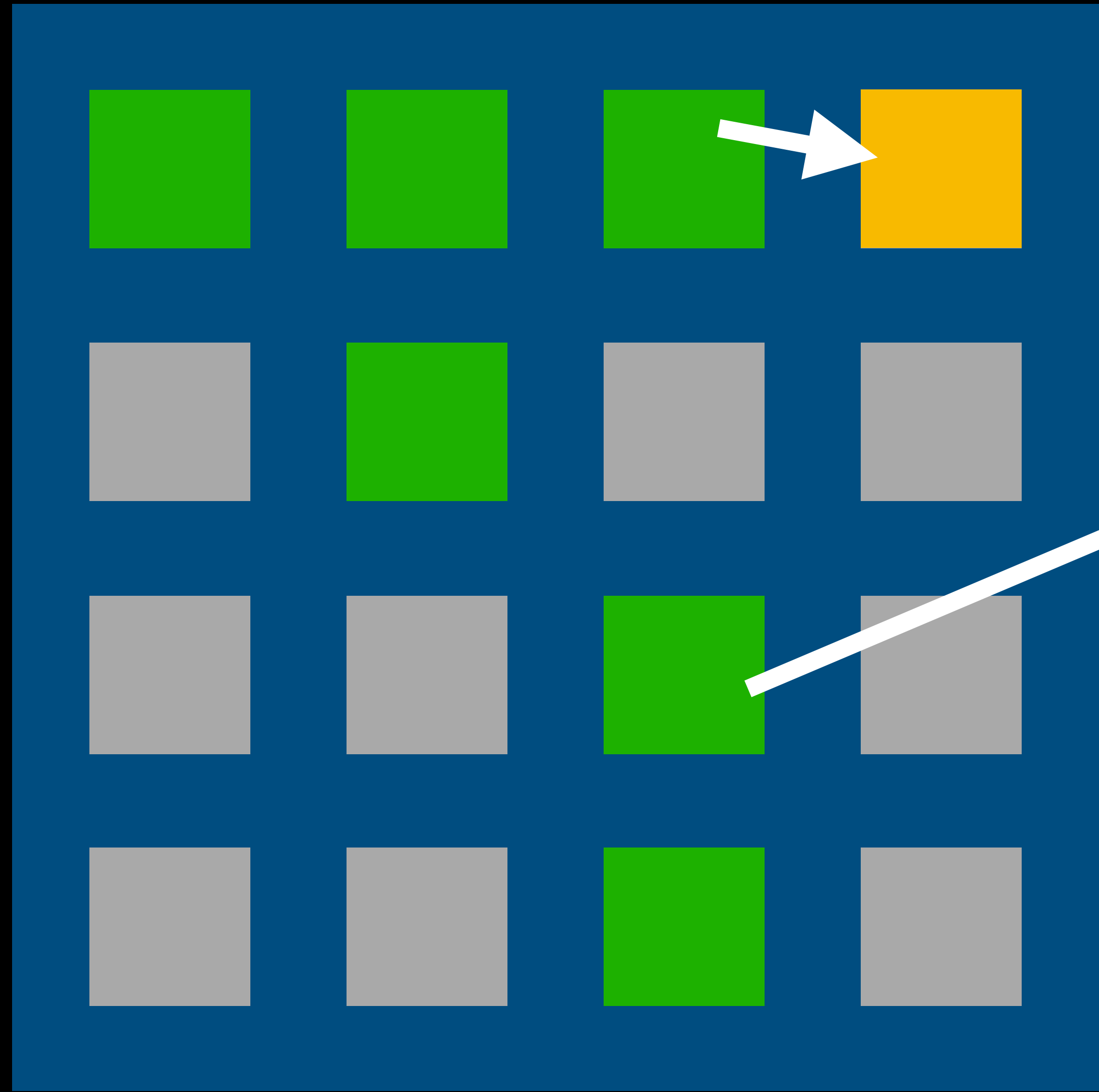
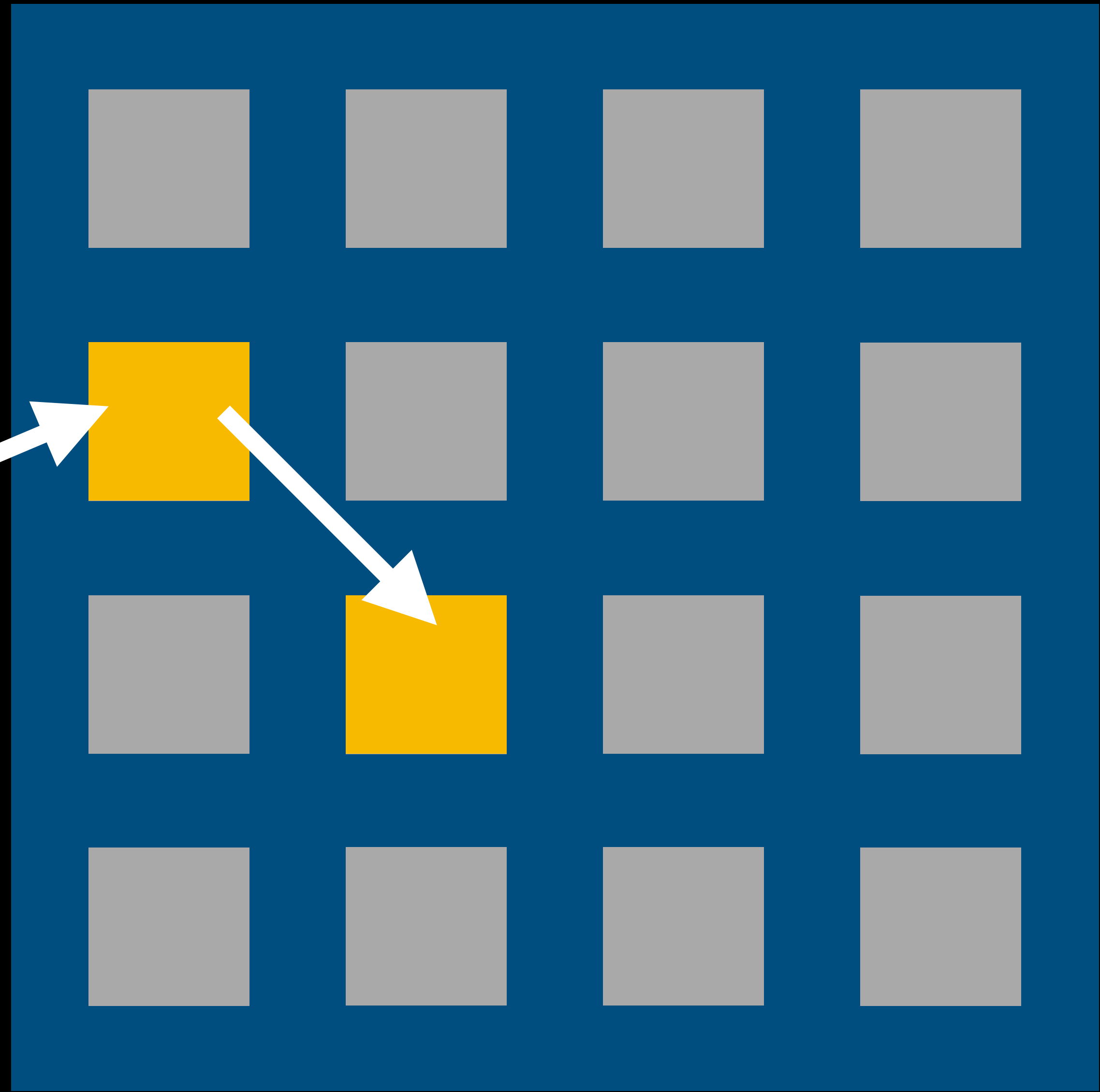


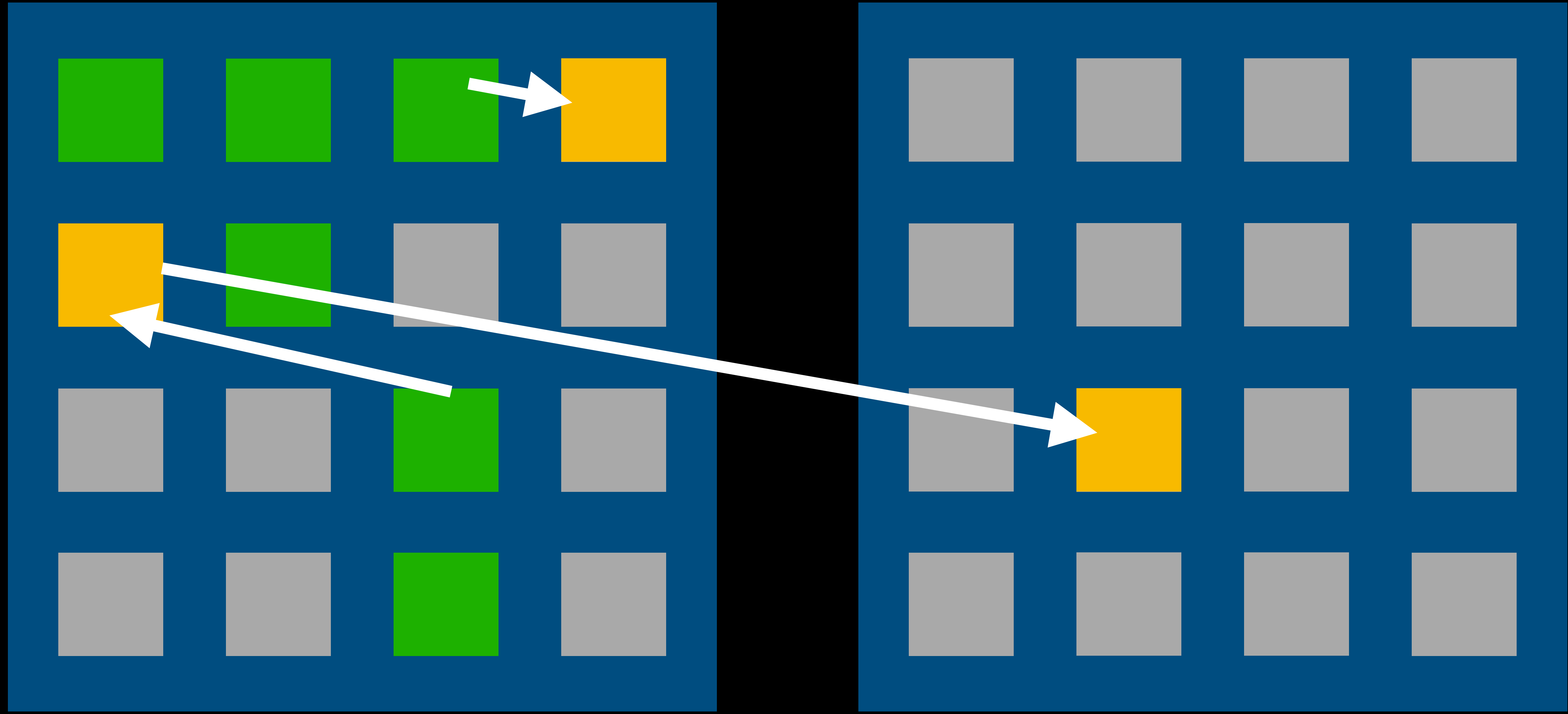
¡Compactación!

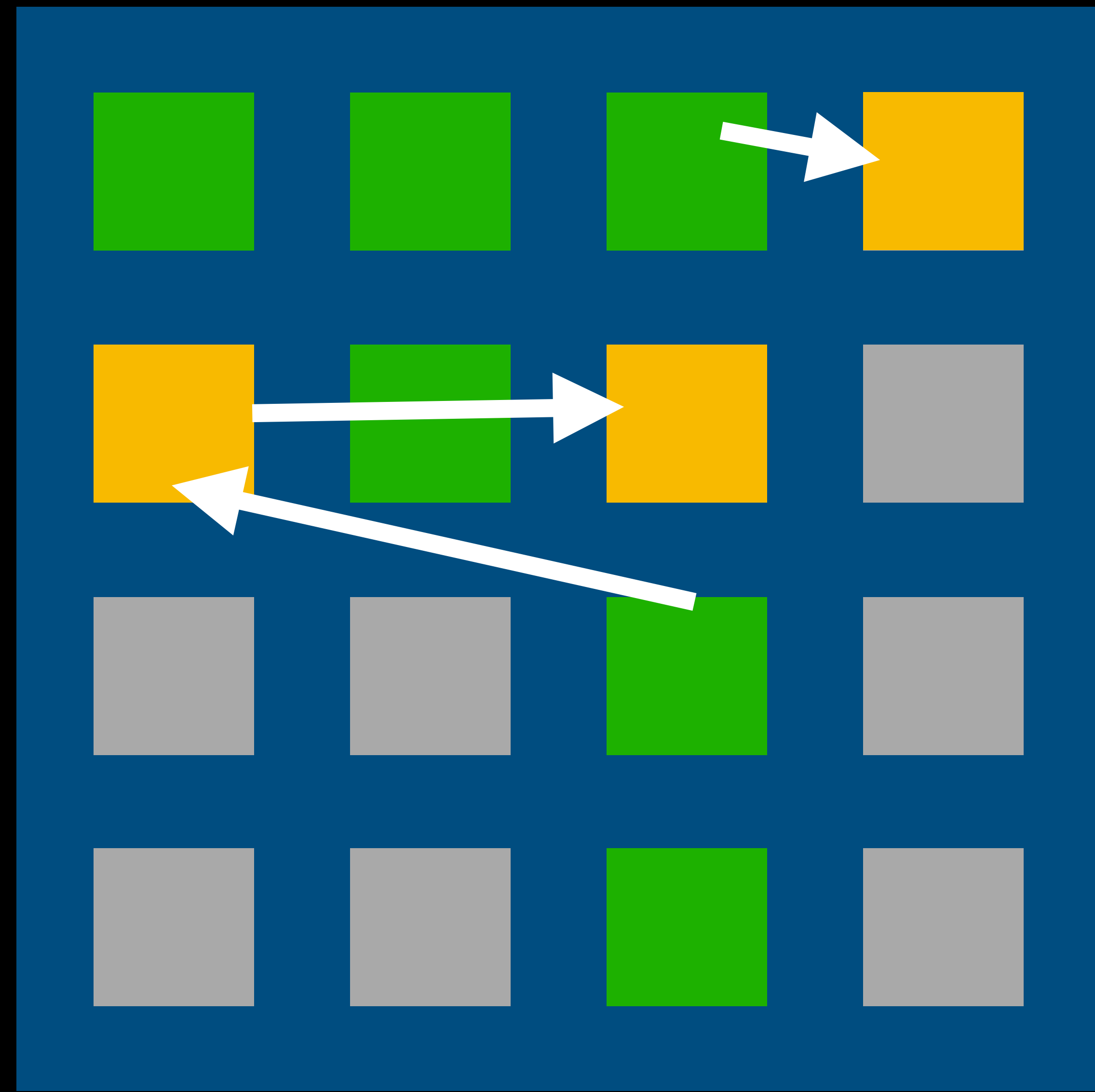
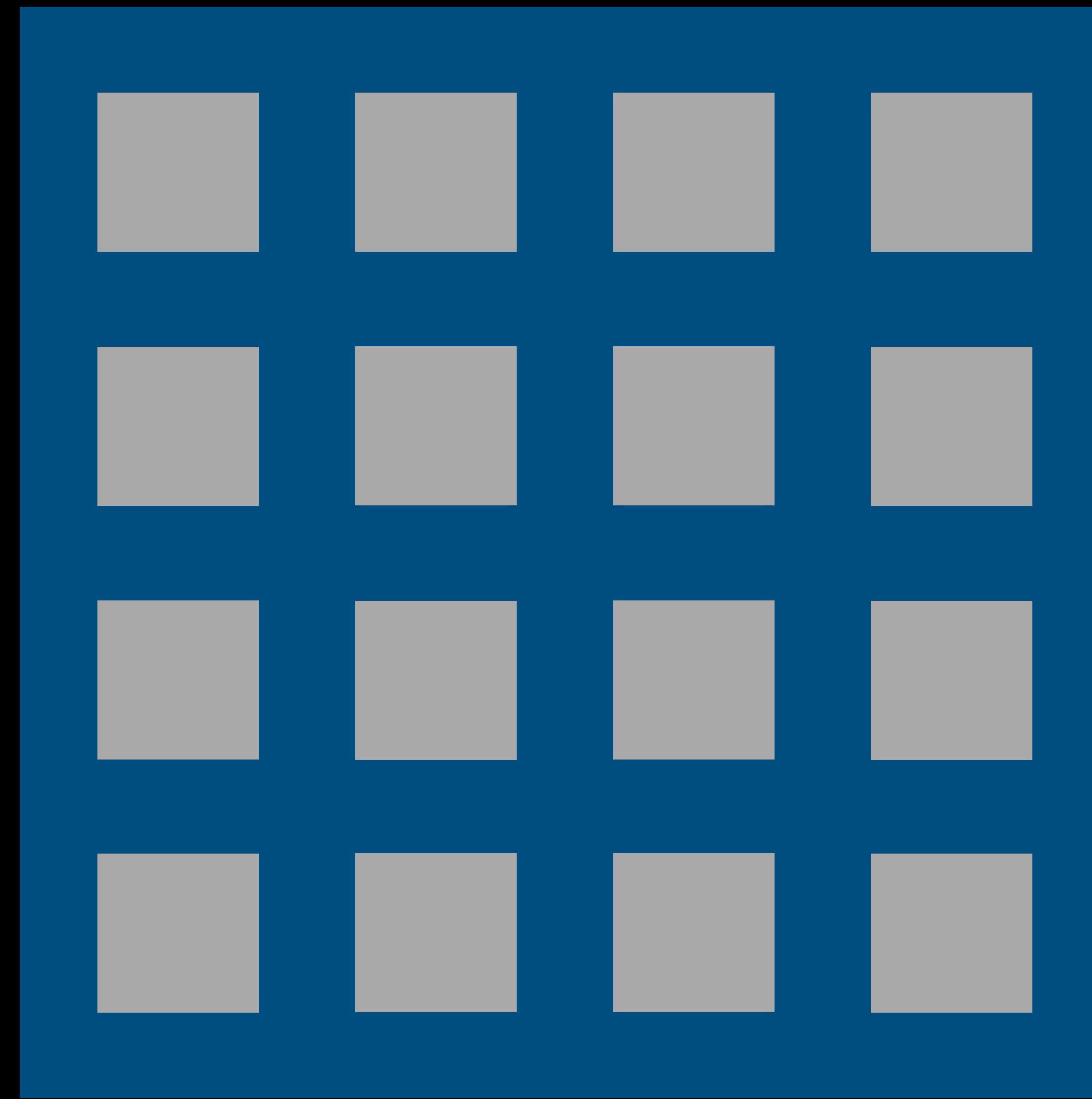
Compaction In Action

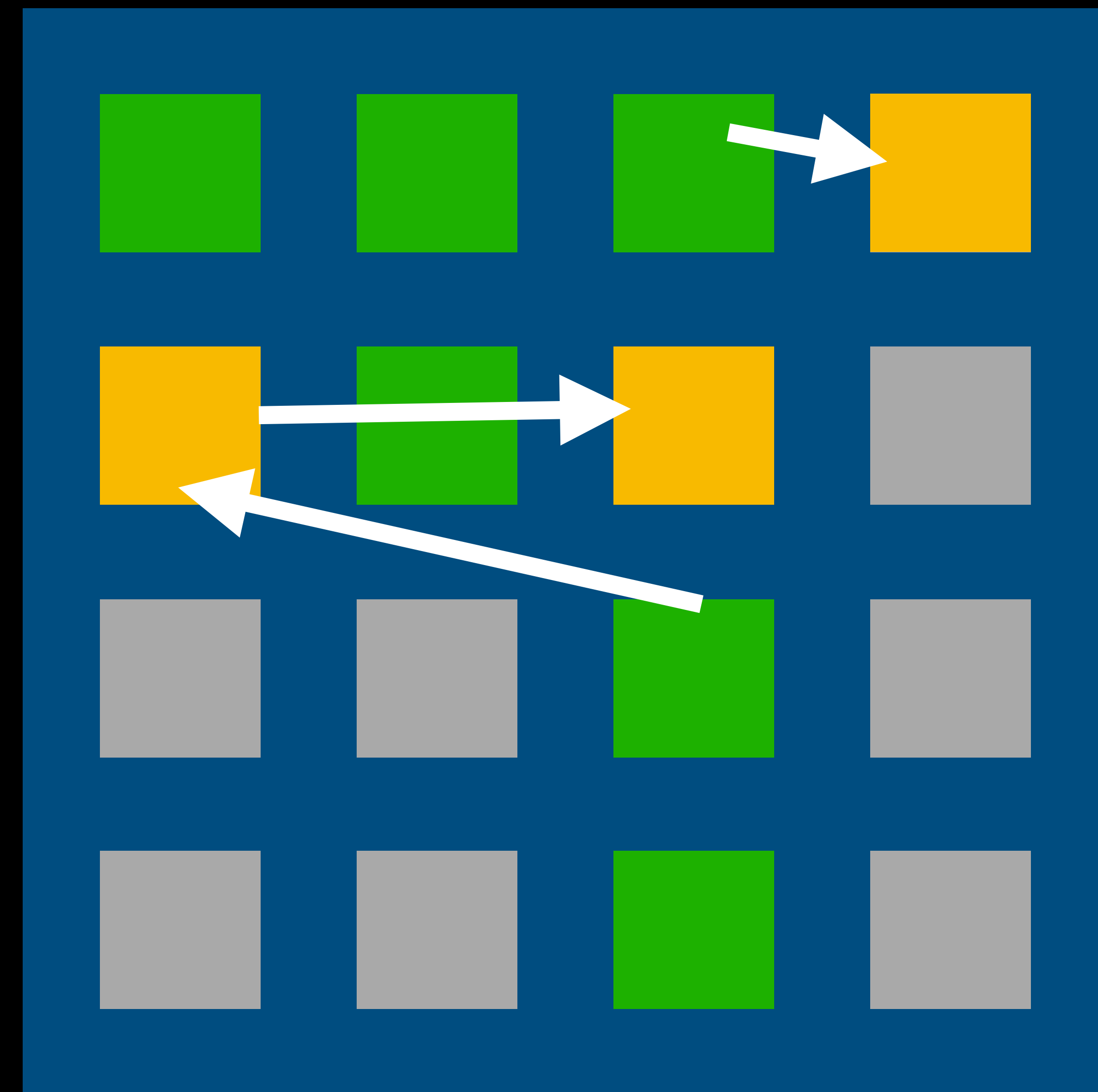


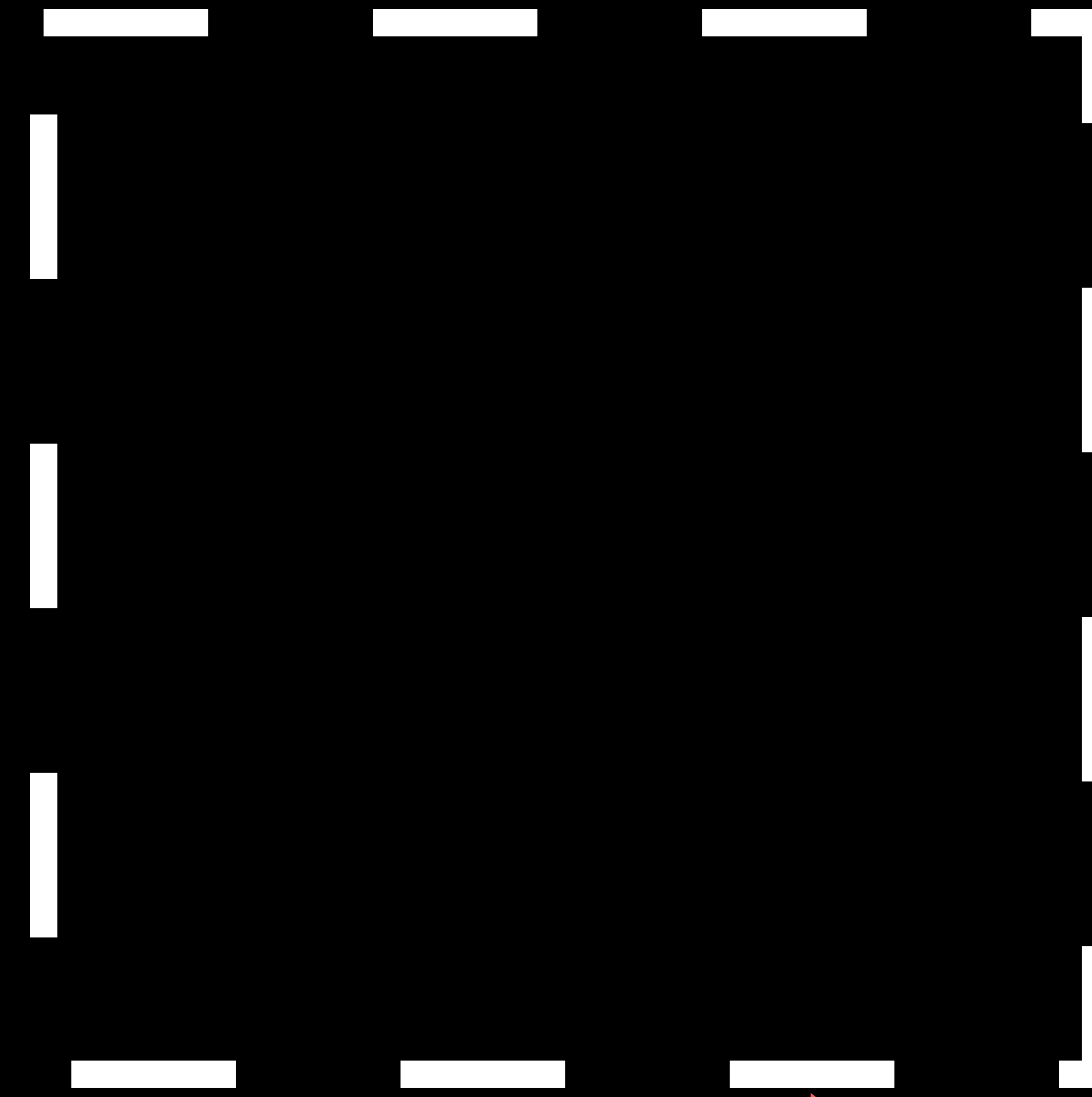








Page returned
to the OS

```
$ clang++ -o yolo main.cc
```

```
$ clang++ -o yolo main.cc
$ strip yolo
```

```
$ clang++ -o yolo main.cc
$ strip yolo
$ ./yolo
```

No way to precisely distinguish pointers from integers

```
$ clang++ -o yolo main.cc
$ strip yolo
$ ./yolo
```

0xDEADC000

No way to precisely distinguish pointers from integers

```
$ clang++ -o yolo main.cc
$ strip yolo
$ ./yolo
```

0xDEADC000

No way to precisely distinguish pointers from integers

0xBEEFC000

0xDEADC000

```
$ clang++ -o yolo main.cc
$ strip yolo
$ ./yolo
```

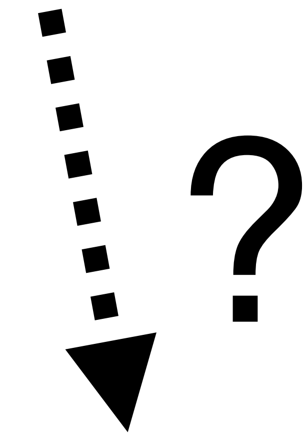
0xDEADC000

No way to precisely distinguish pointers from integers

0xBEEFC000

0xDEADC000

```
$ clang++ -o yolo main.cc
$ strip yolo
$ ./yolo
```



0xDEADC000

No way to precisely distinguish pointers from integers

0xBEEFC000

```
union tiny
{
 int * ptr;
 uintptr_t flag;
};
tiny x;

// initialize
x.ptr = new int;

// set flag true
x.flag |= 1;
```

```
union tiny
{
 int * ptr;
 uintptr_t flag;
};
tiny x;

// initialize
x.ptr = new int;

// set flag true
x.flag |= 1;
```

0xDEADC000

0xBEEFC000

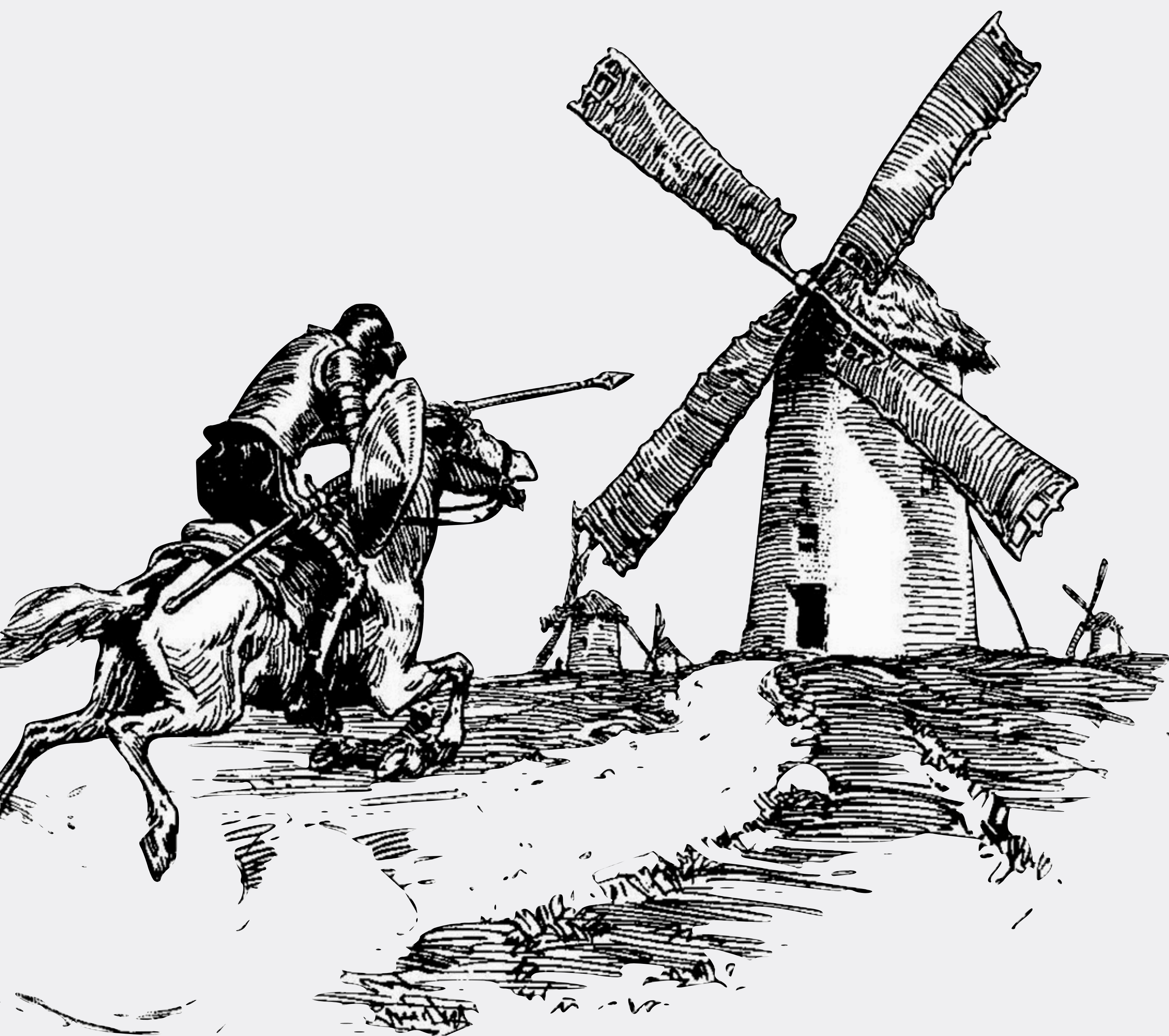
```
union tiny
{
 int * ptr;
 uintptr_t flag;
};
tiny x;

// initialize
x.ptr = new int;

// set flag true
x.flag |= 1;
```

0xDEADC000

0xBEEFC000



MESH

Compaction without Relocation for C/C++

No code changes

No recompilation

`LD_PRELOAD` and go

17% heap size reduction

< 1% performance overhead

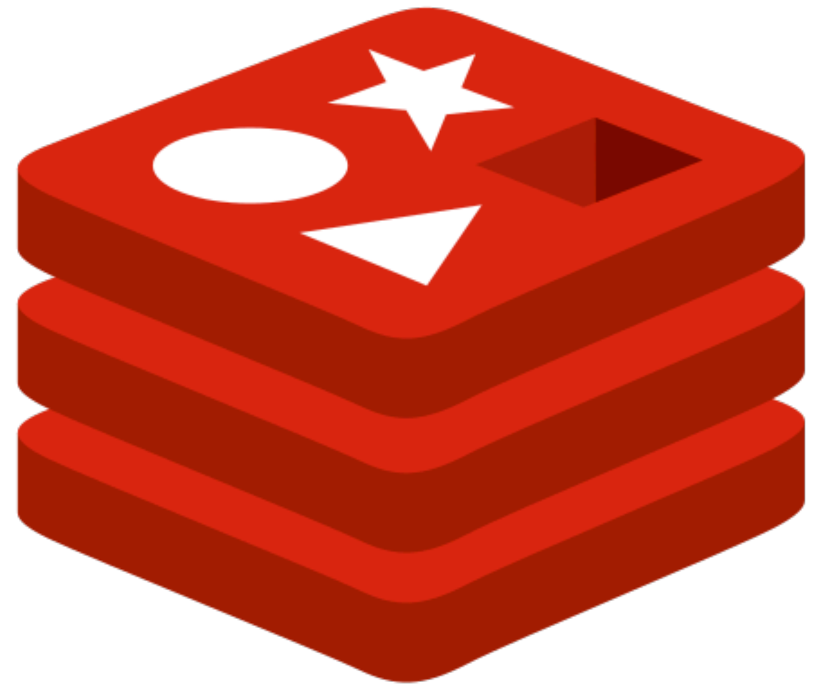
RSS (MiB)

800
600
400
200
0

0
50
100

Time Since Program Start (seconds)

—default jemalloc —Mesh



redis

jemalloc + activedefrag

Mesh

no compaction

RSS (MiB)

300

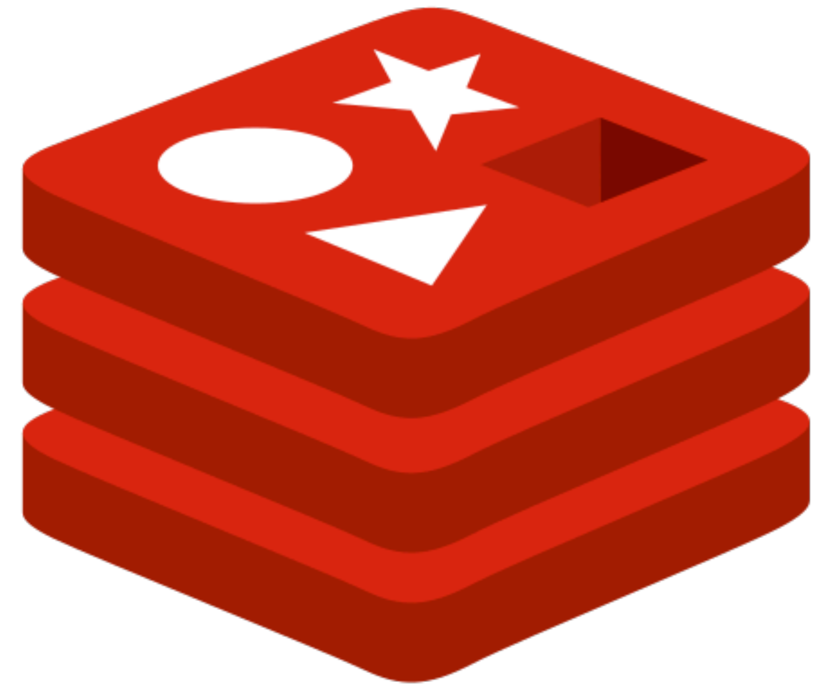
200

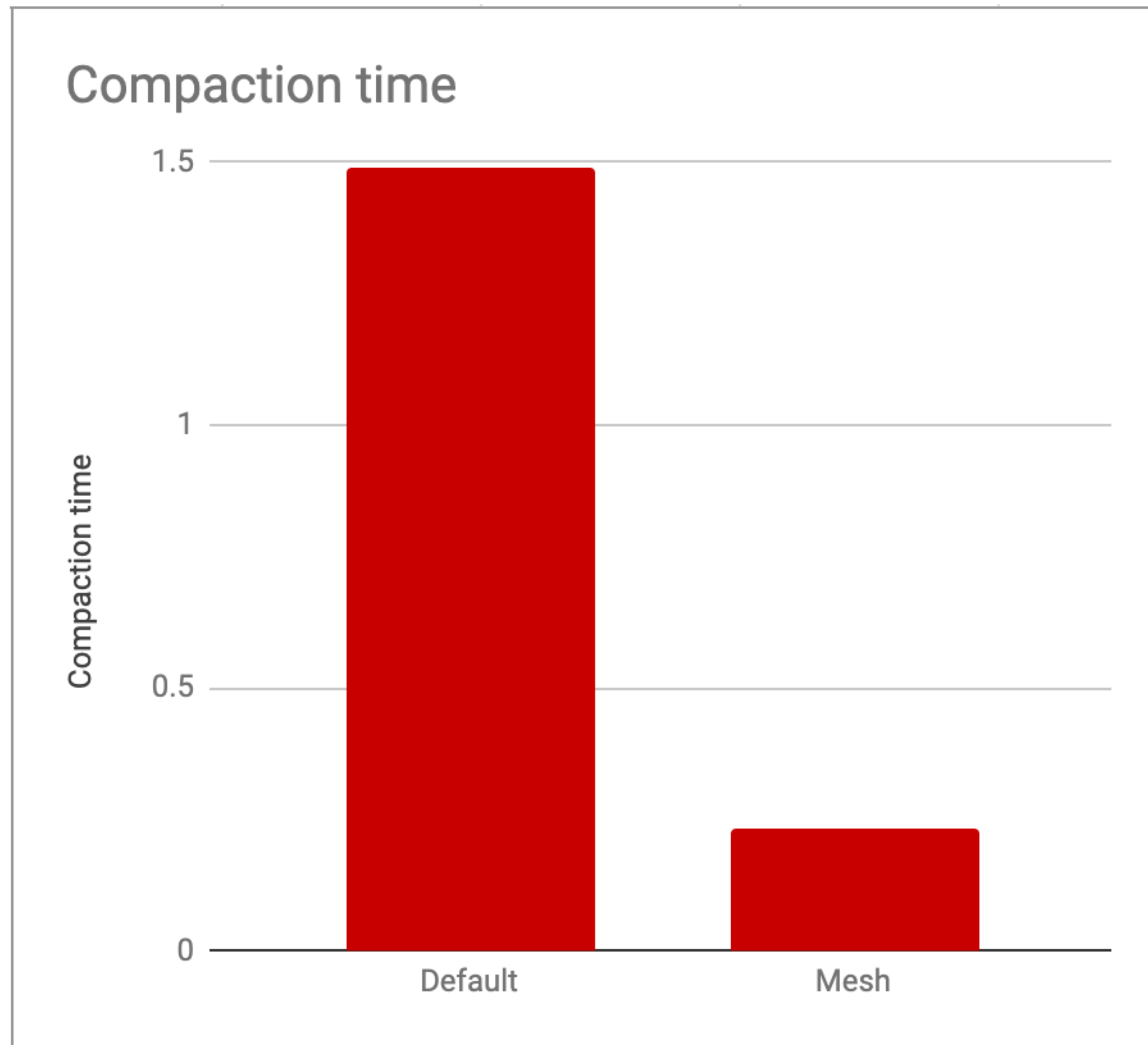
100

0

0 1 2 3 4 5 6

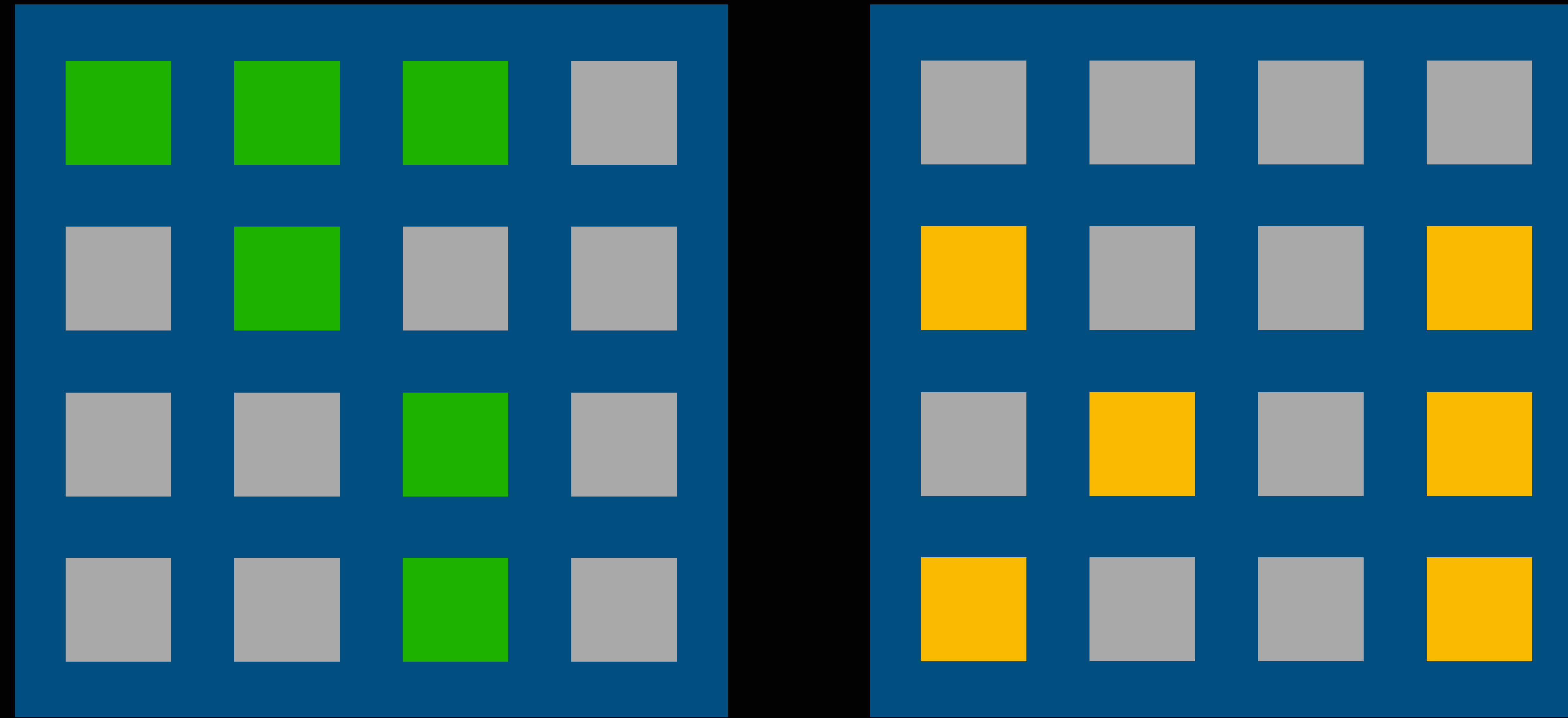
Time Since Program Start (seconds)





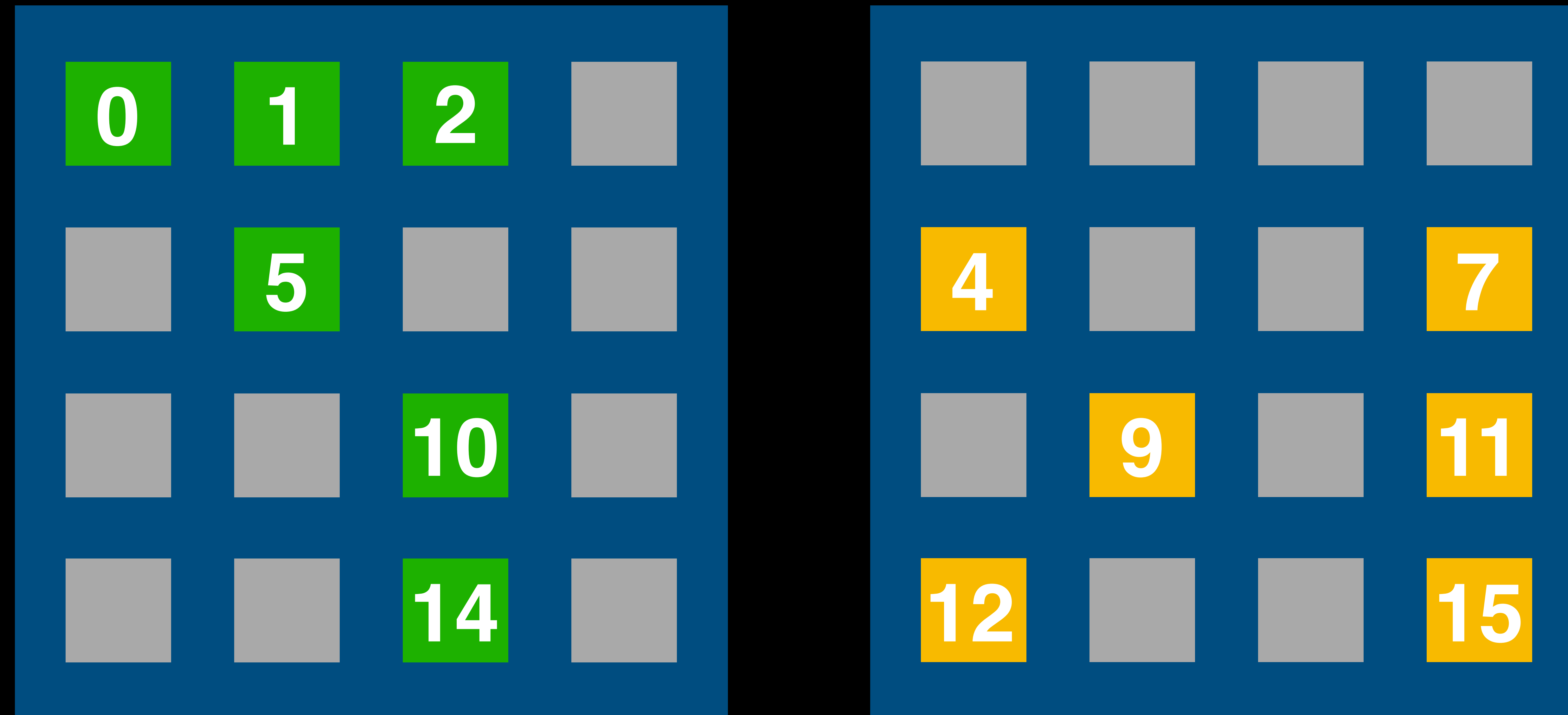
5x reduction in time spent compacting

Meshing: compaction without (virtual!) relocation



Pages **meshable** when they:

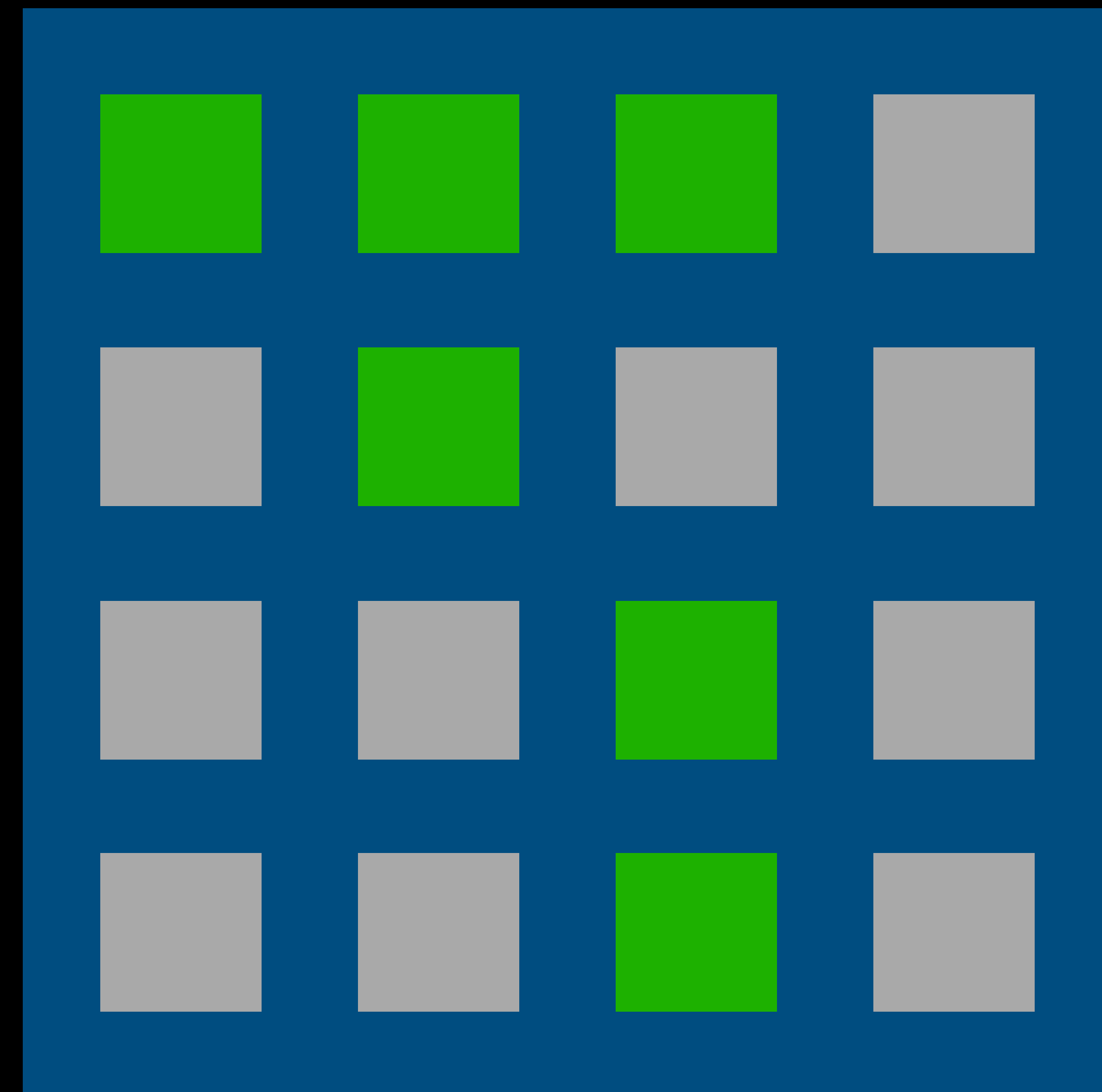
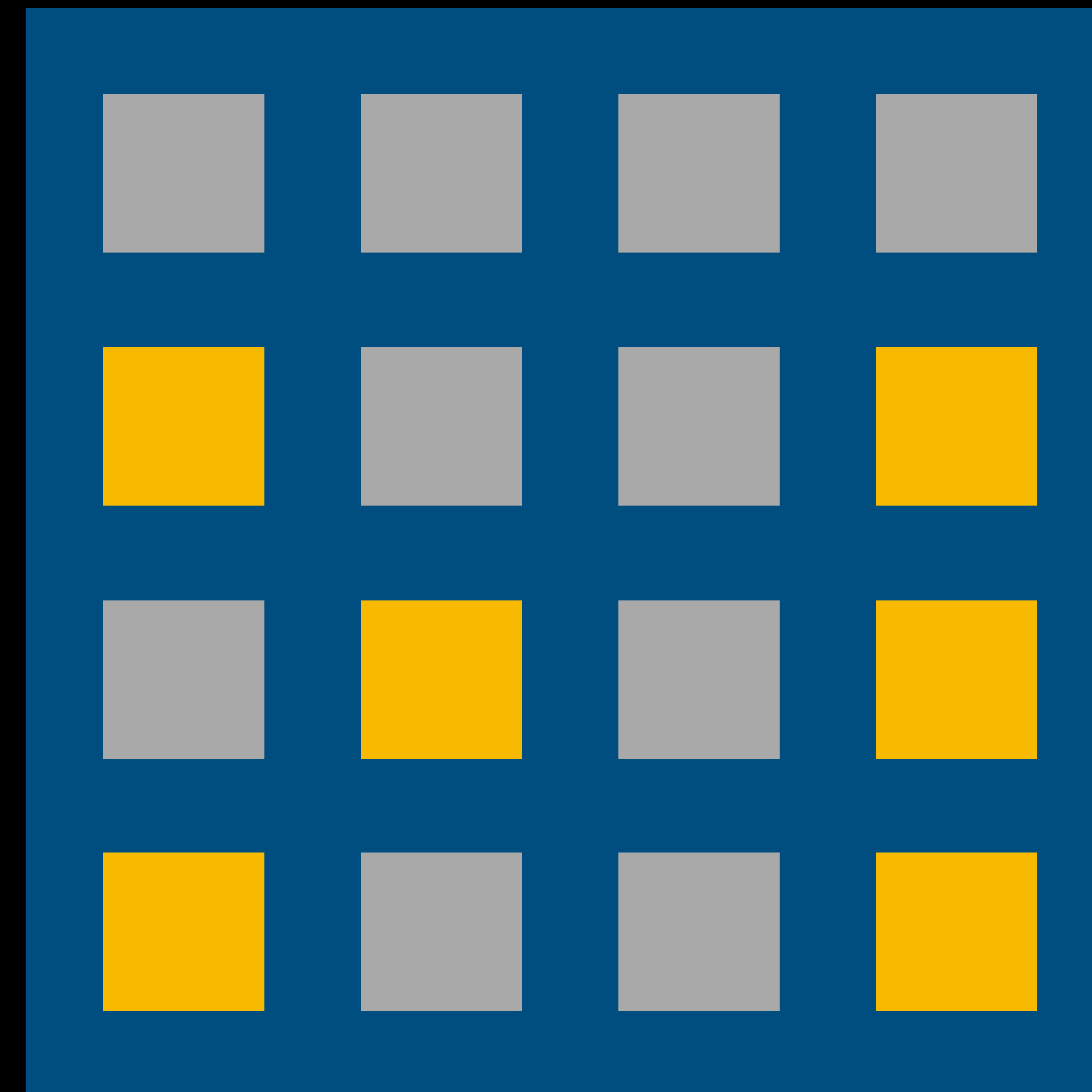
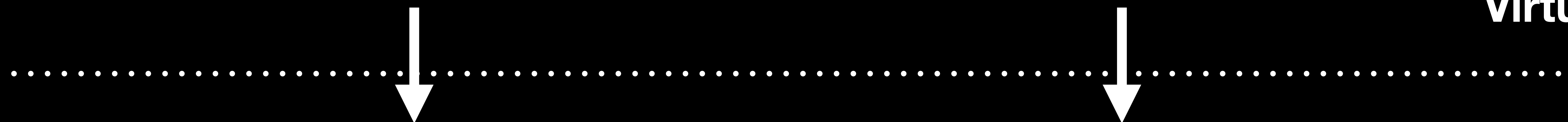
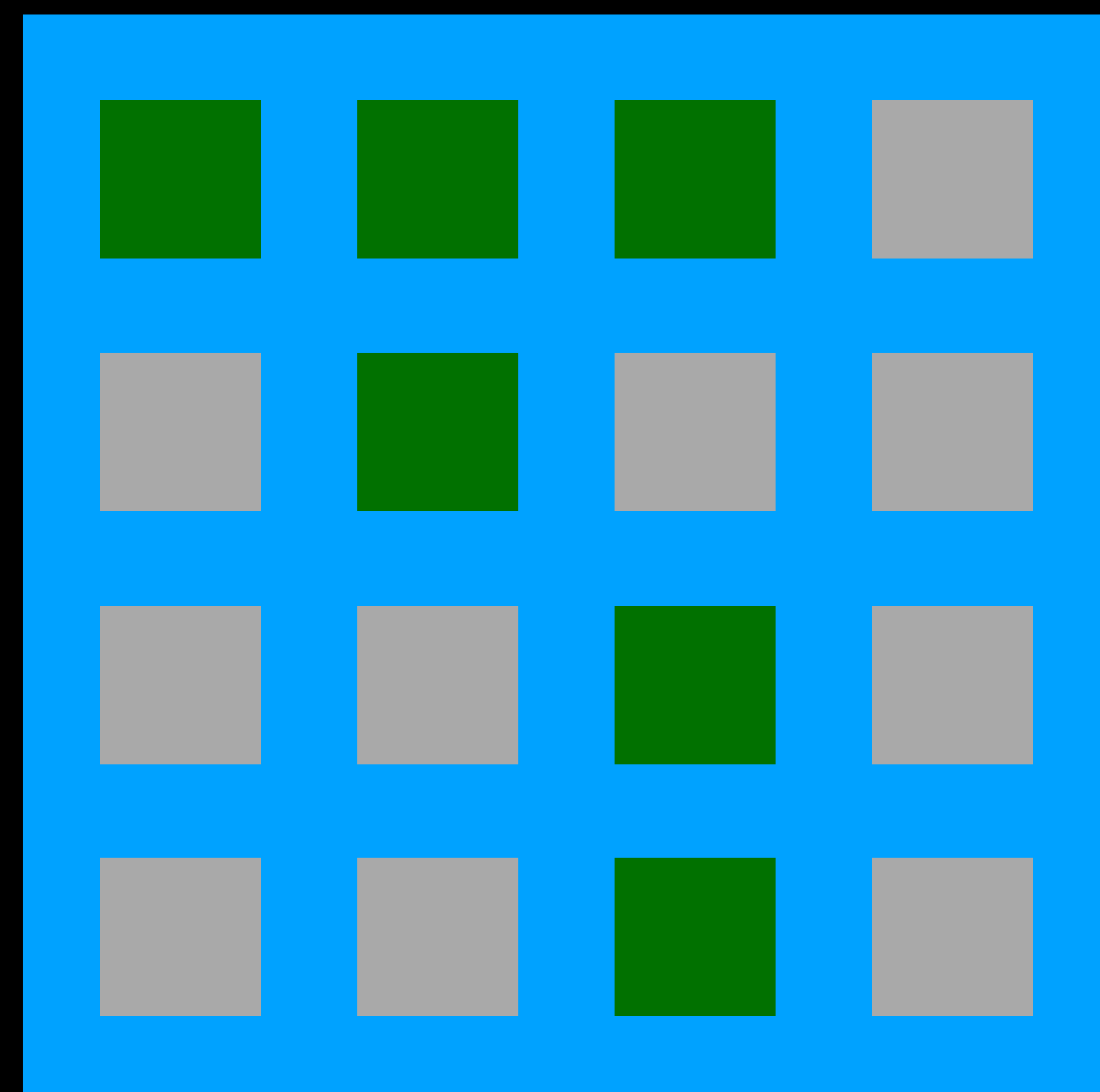
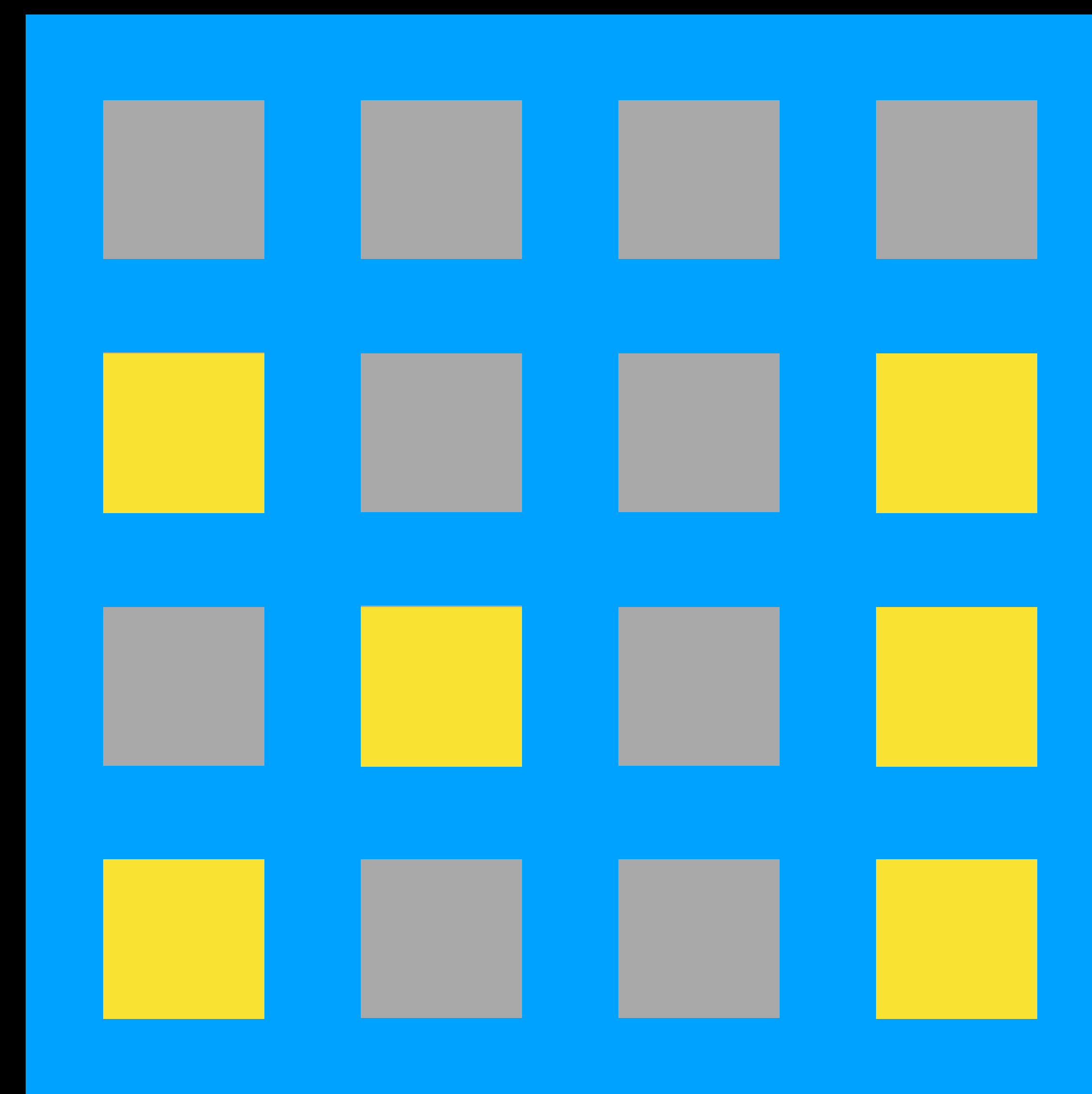
- Hold objects of same size class



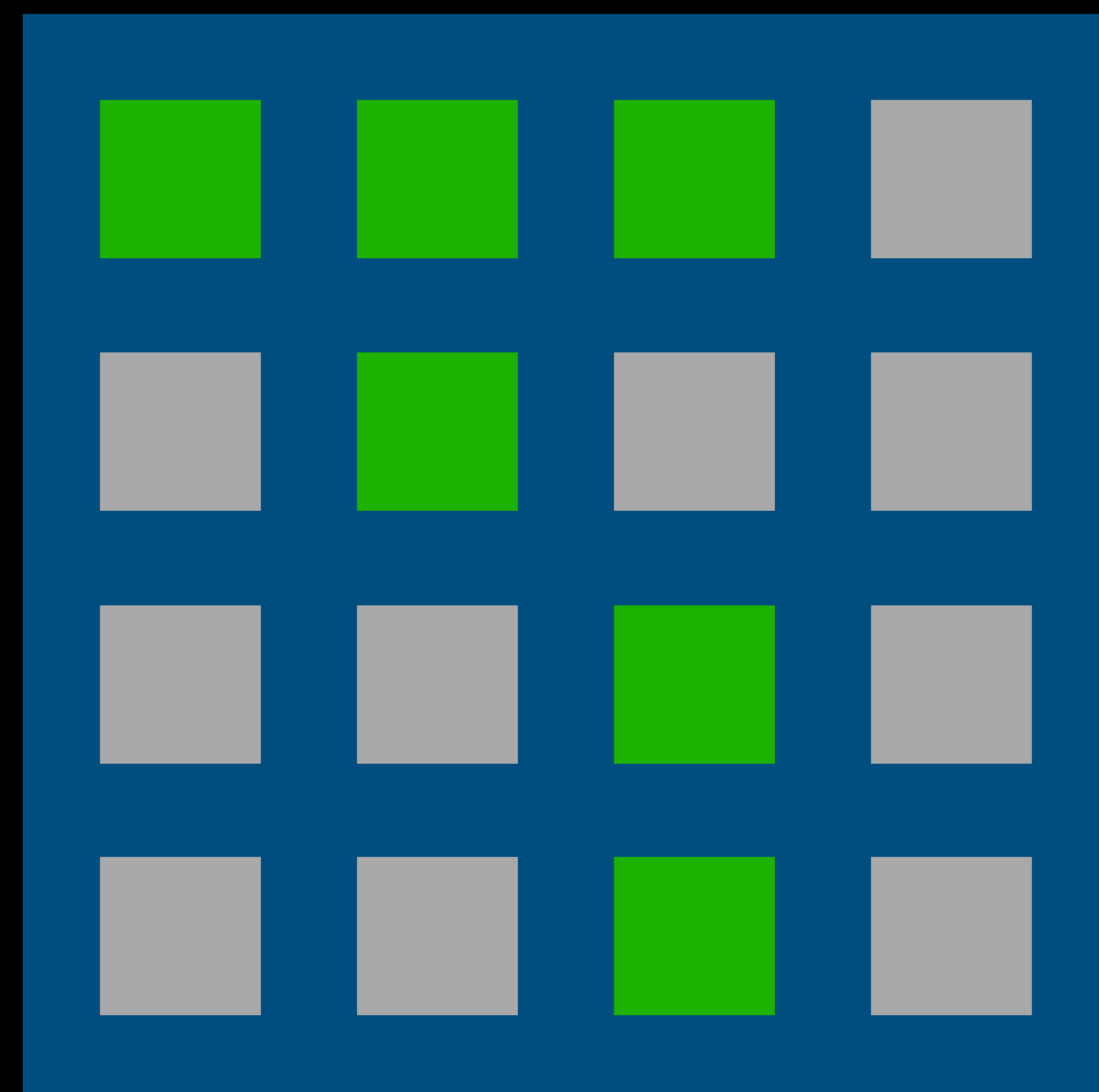
Pages **meshable** when they:

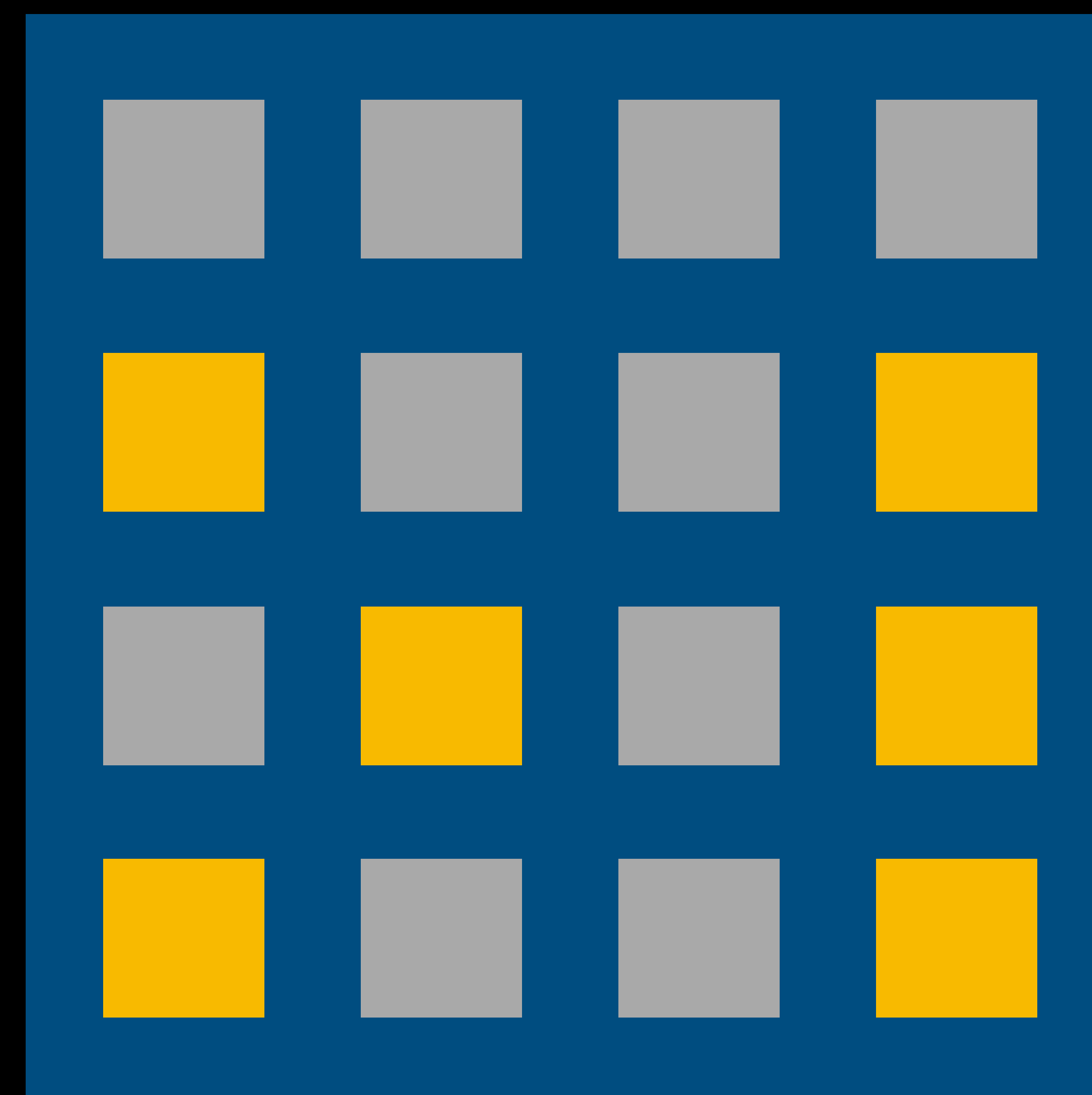
- Hold objects of same size class
- Have non-overlapping object offsets

Meshing



Virtual

Physical



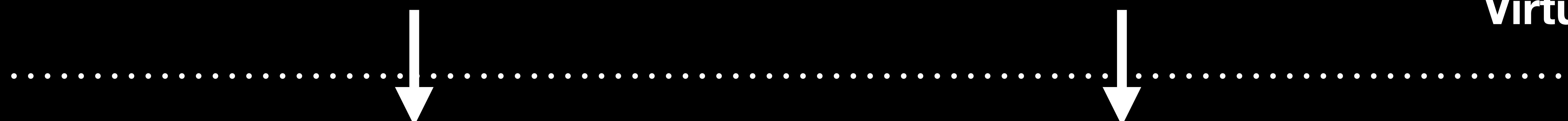
Meshing



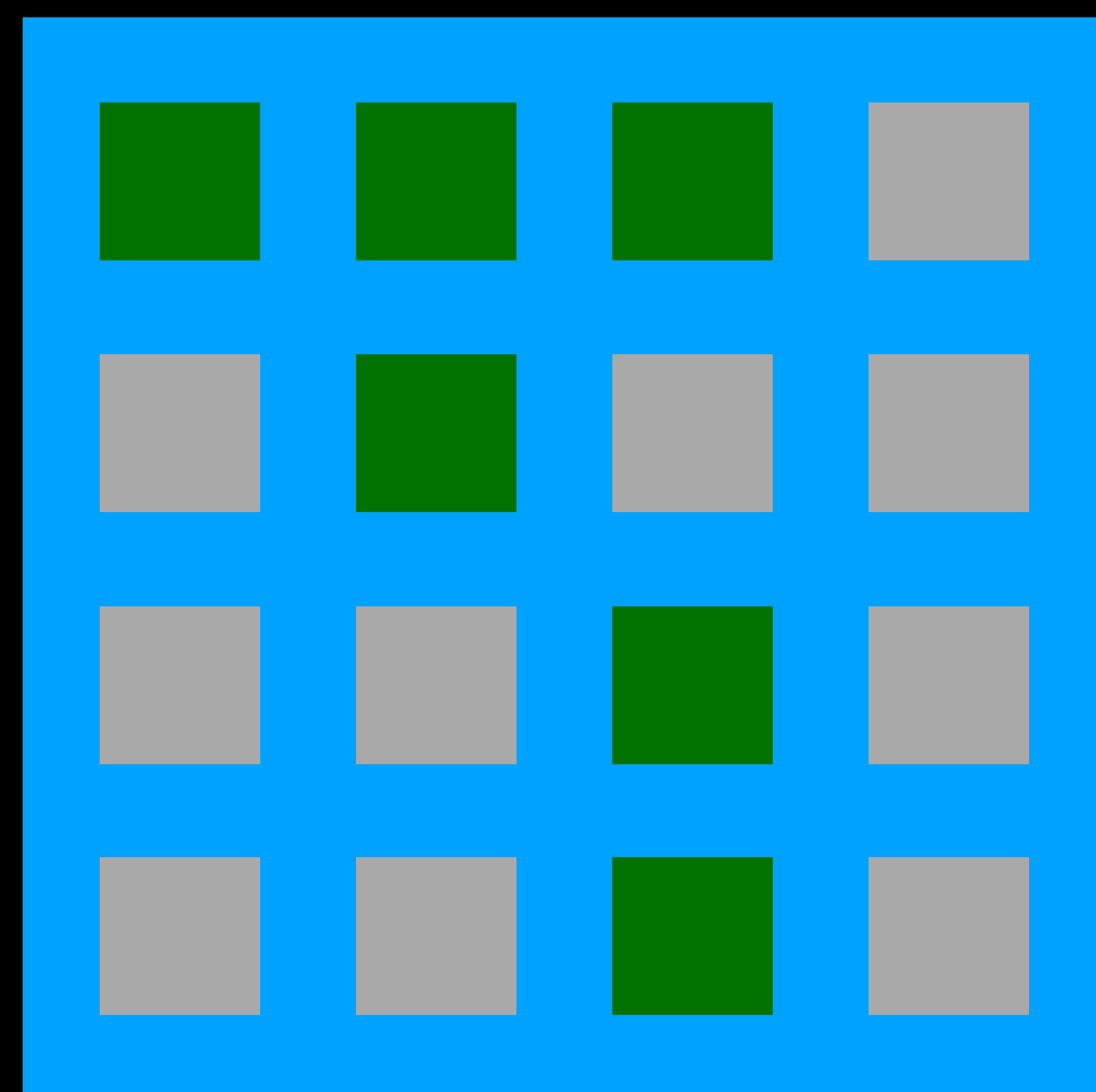


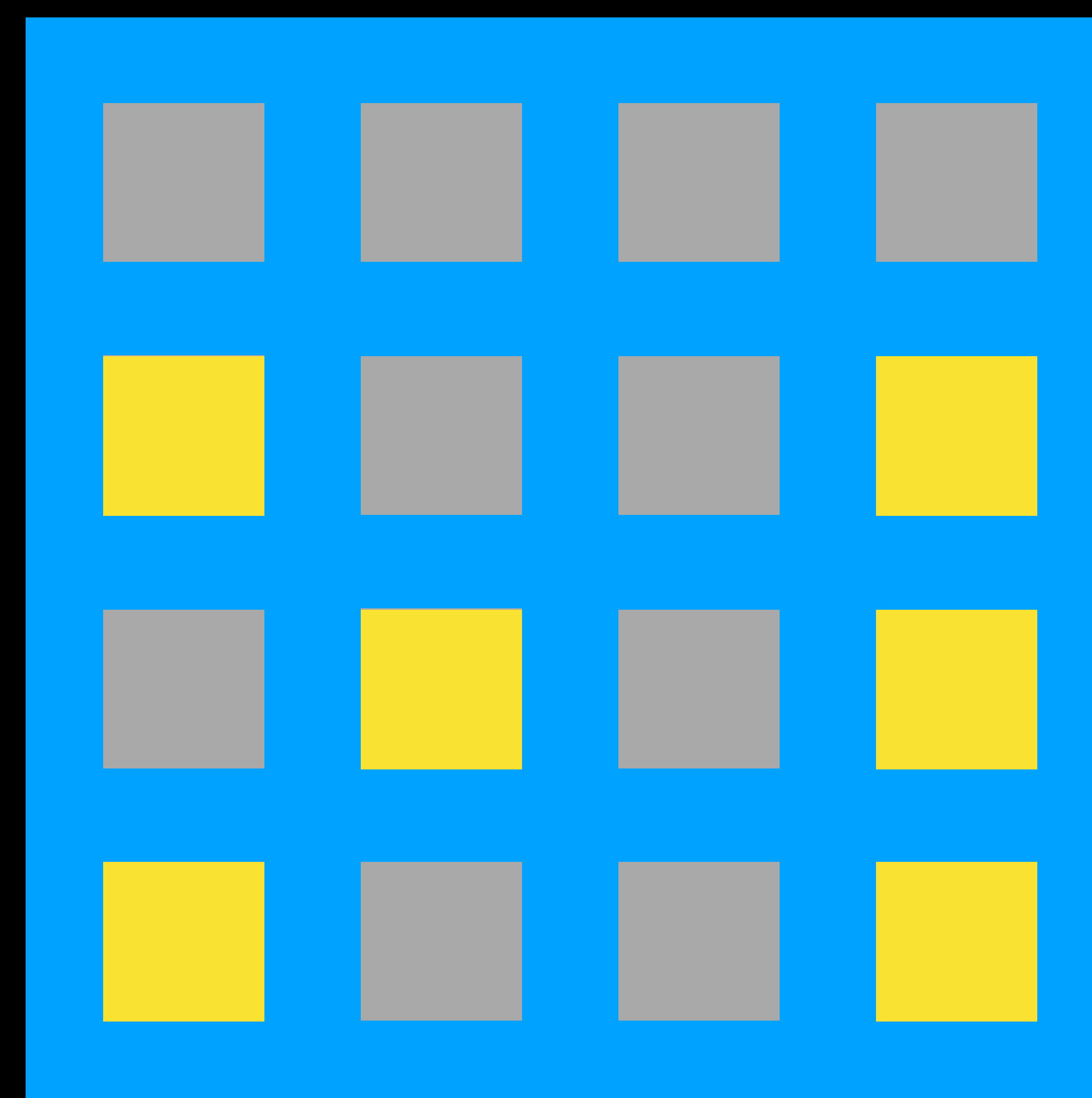
Mark virtual page read-only

Virtual

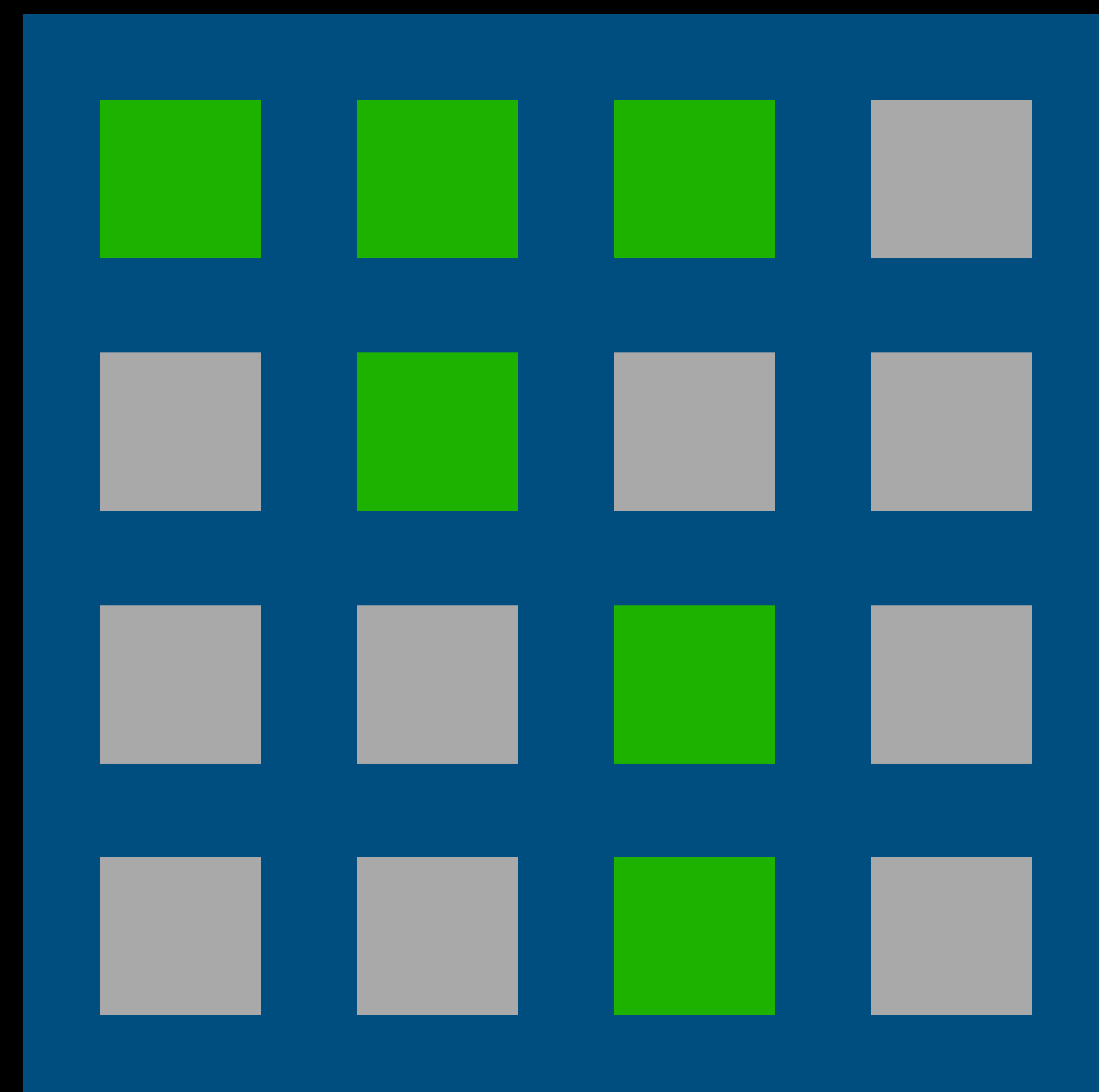
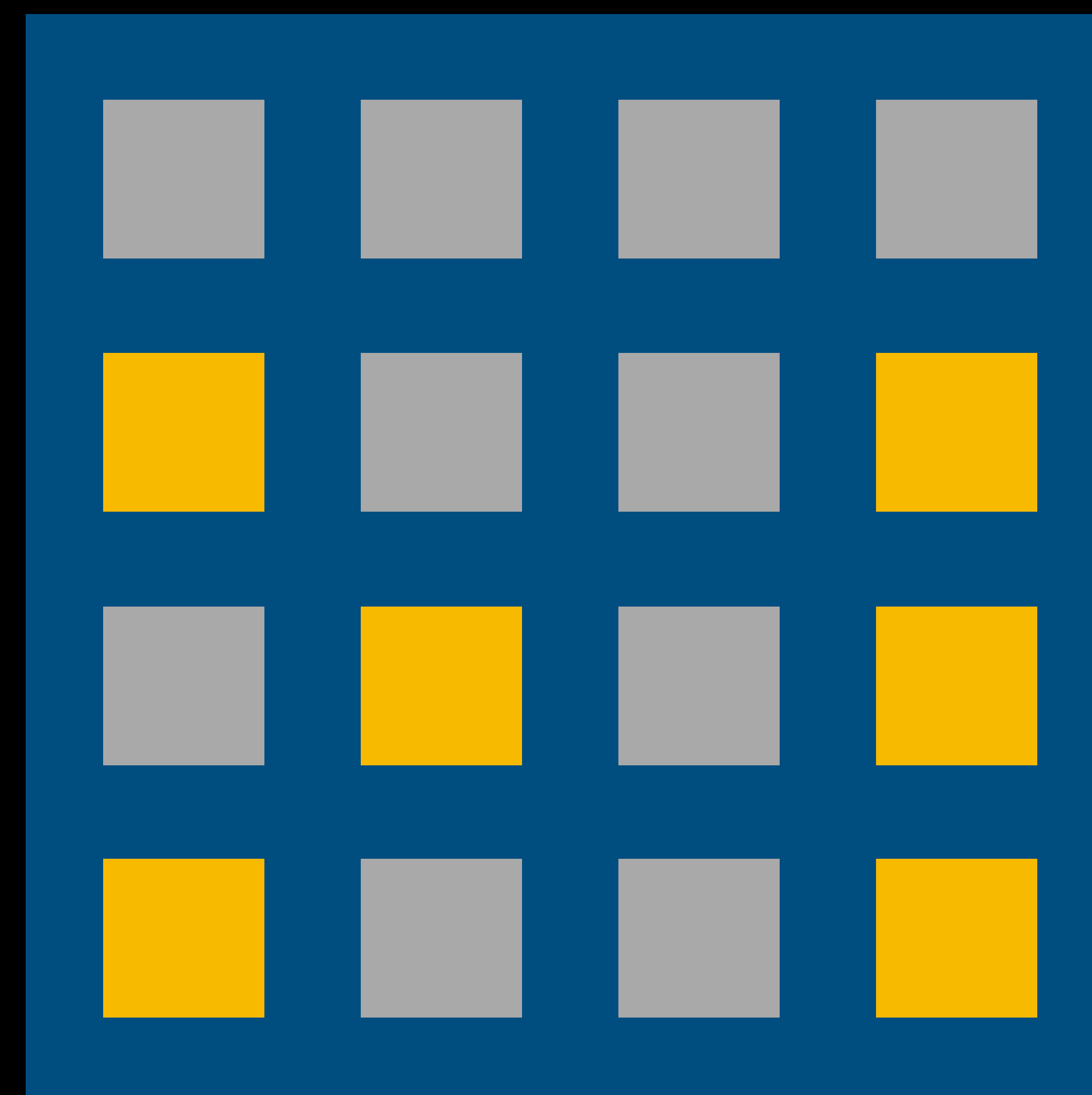


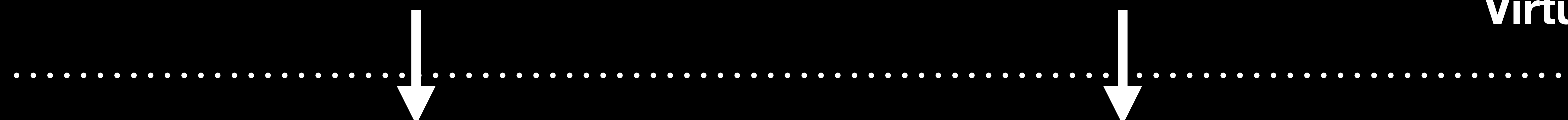
Physical



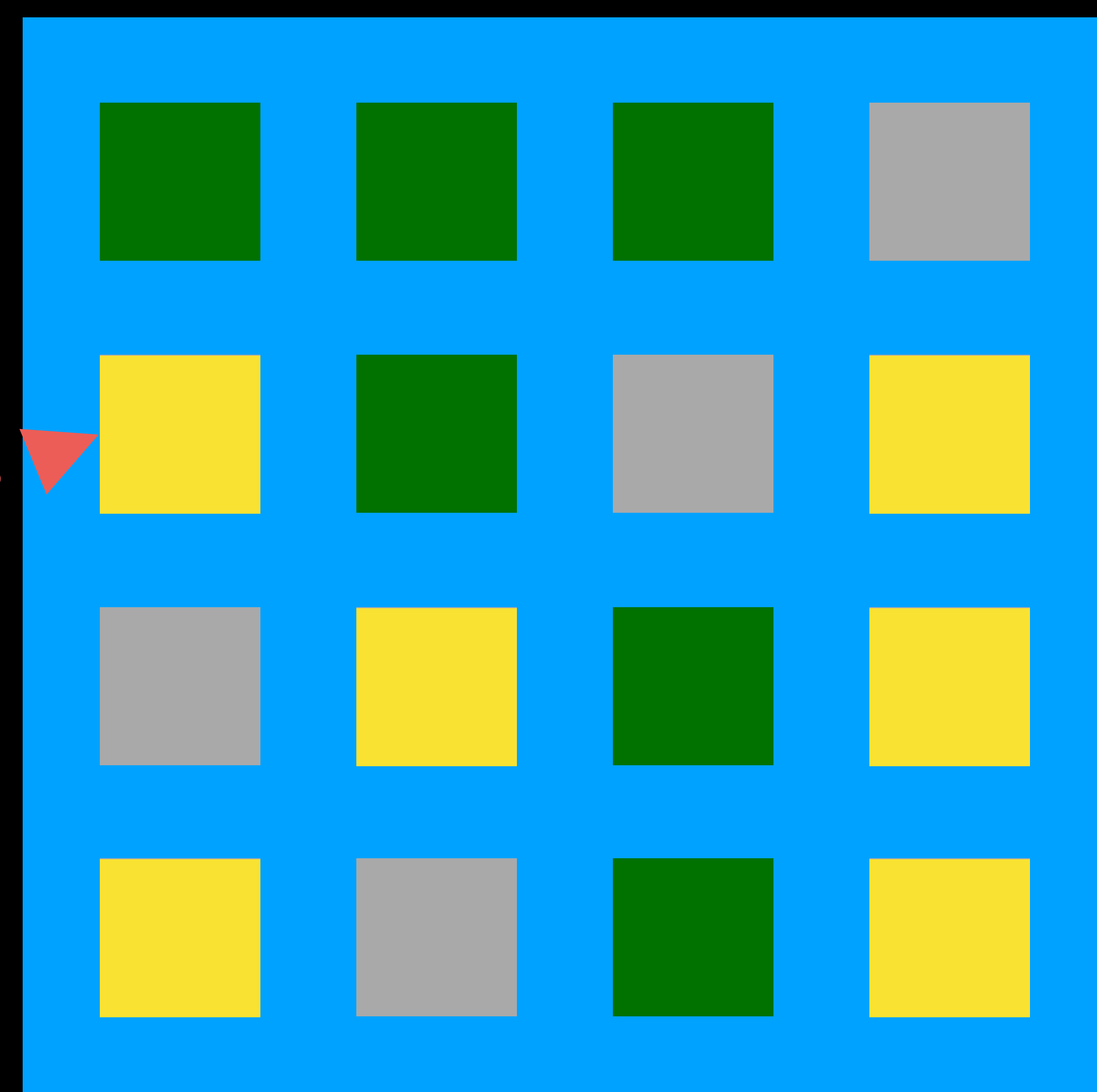
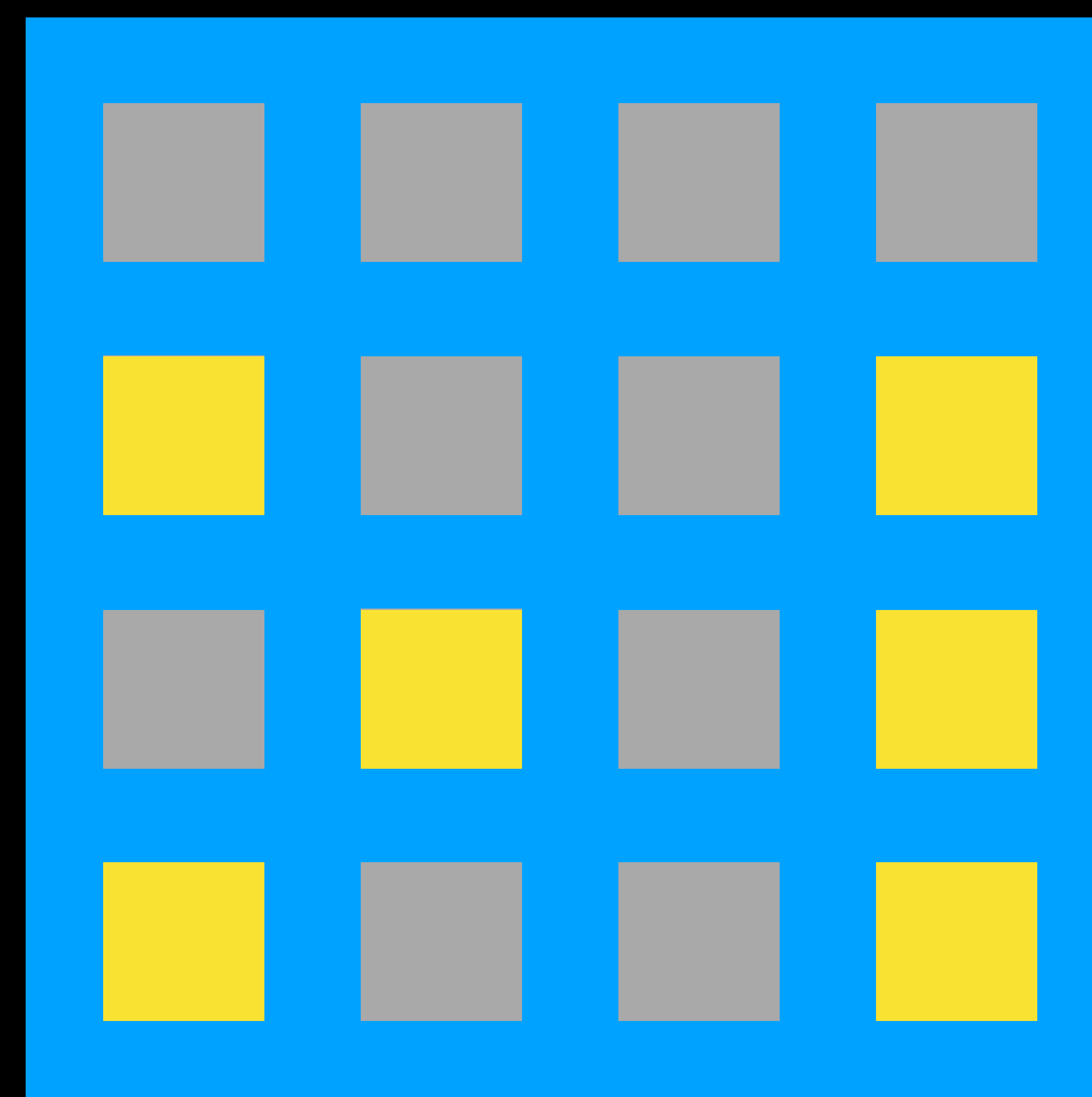


Meshing

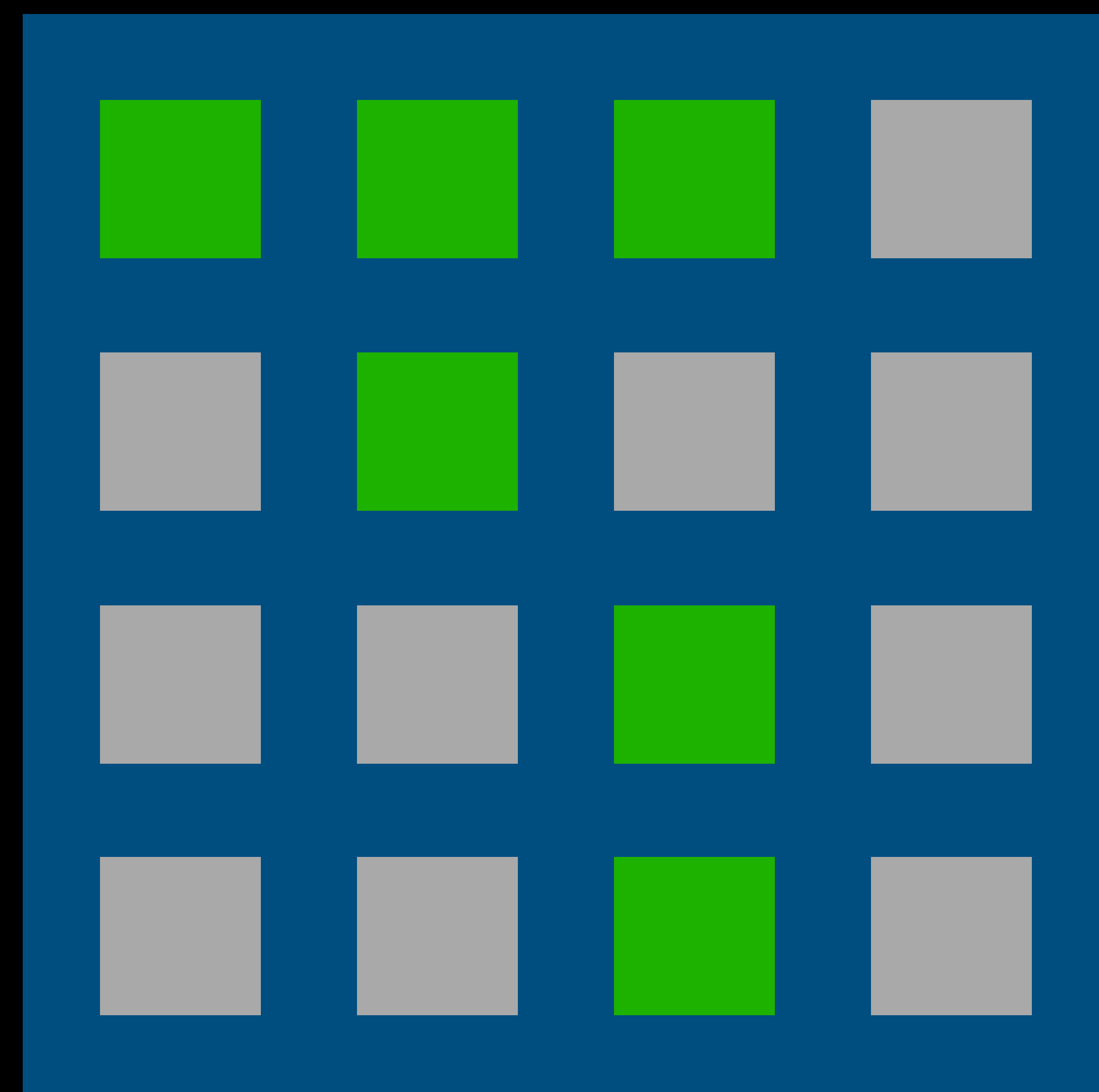
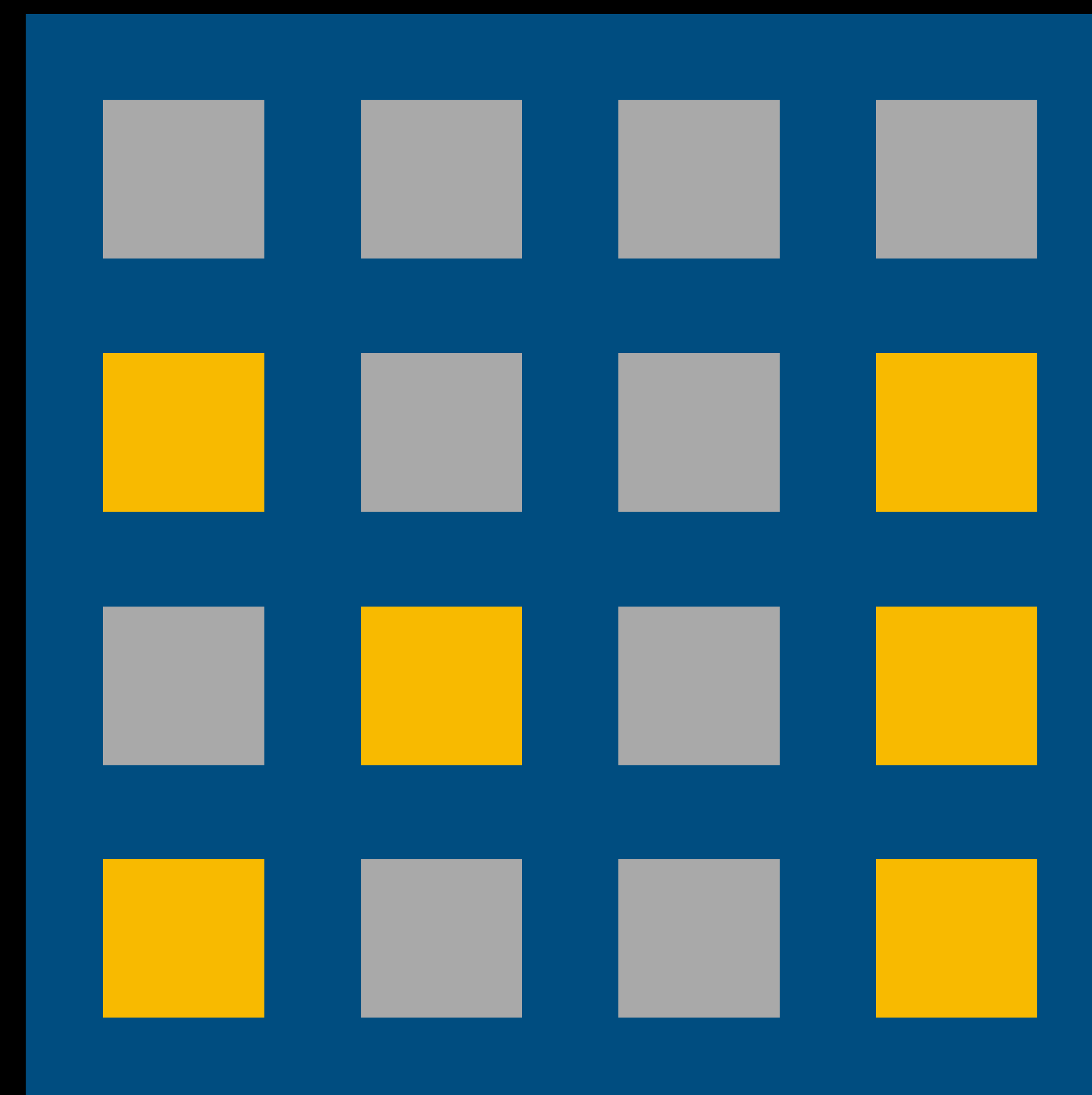
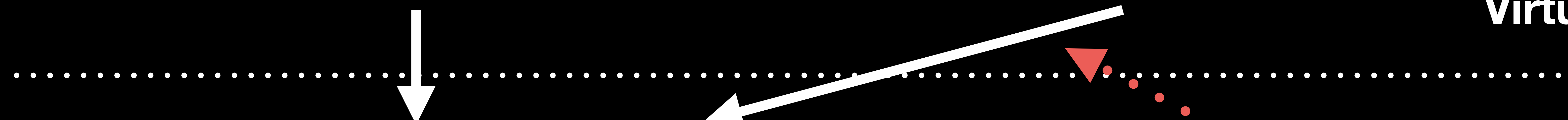
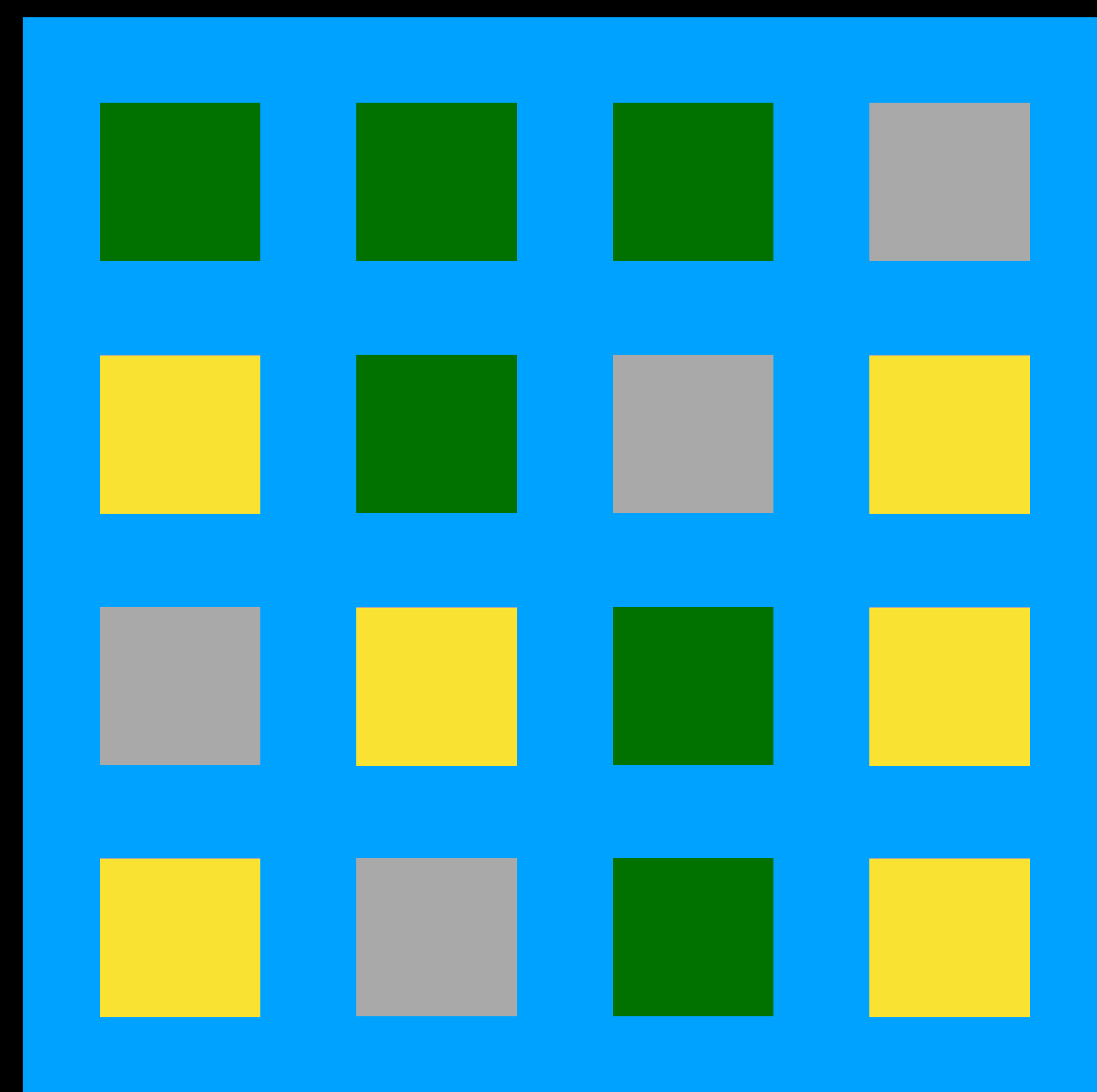
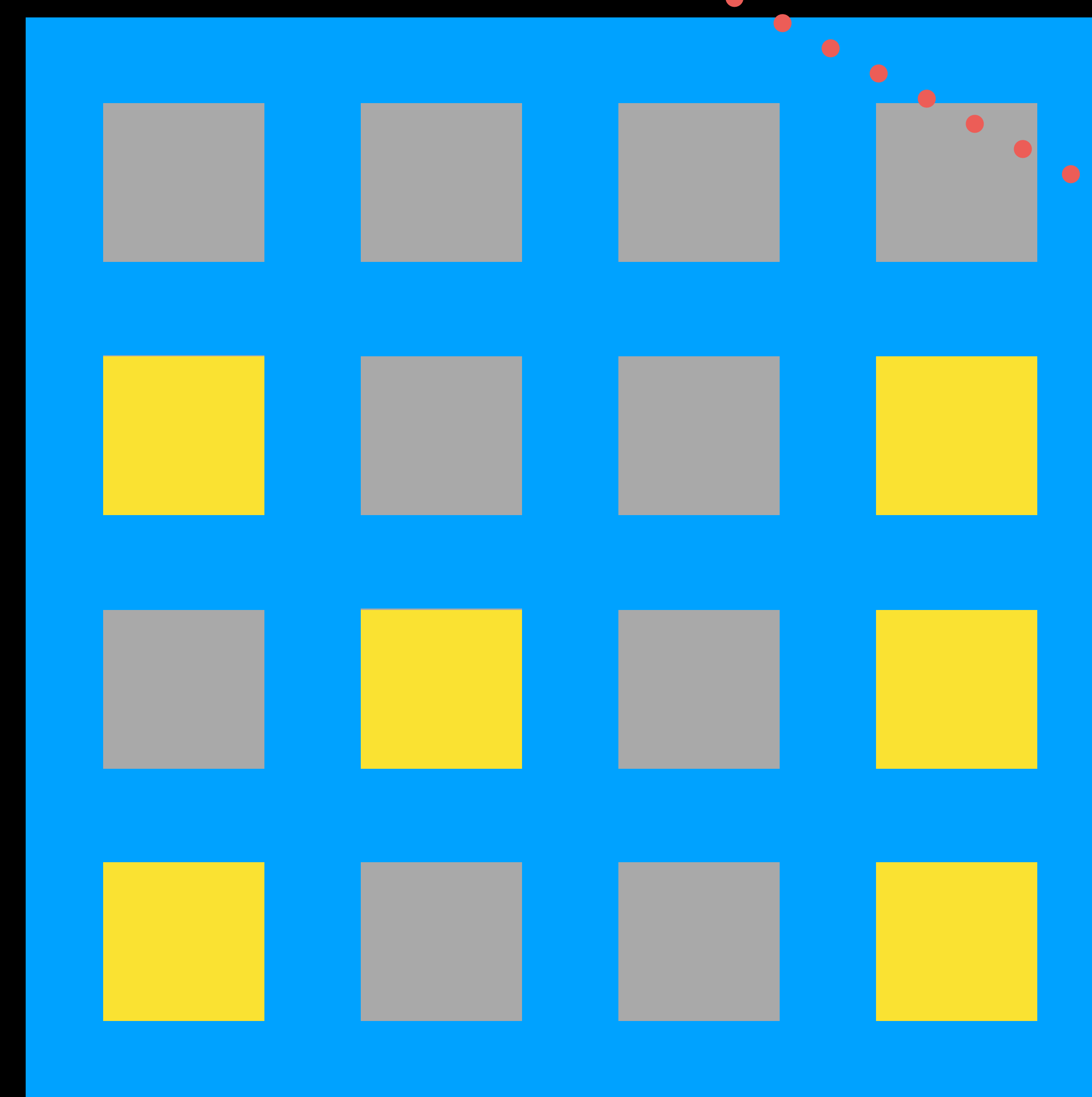





Copy
(maintaining
offsets)

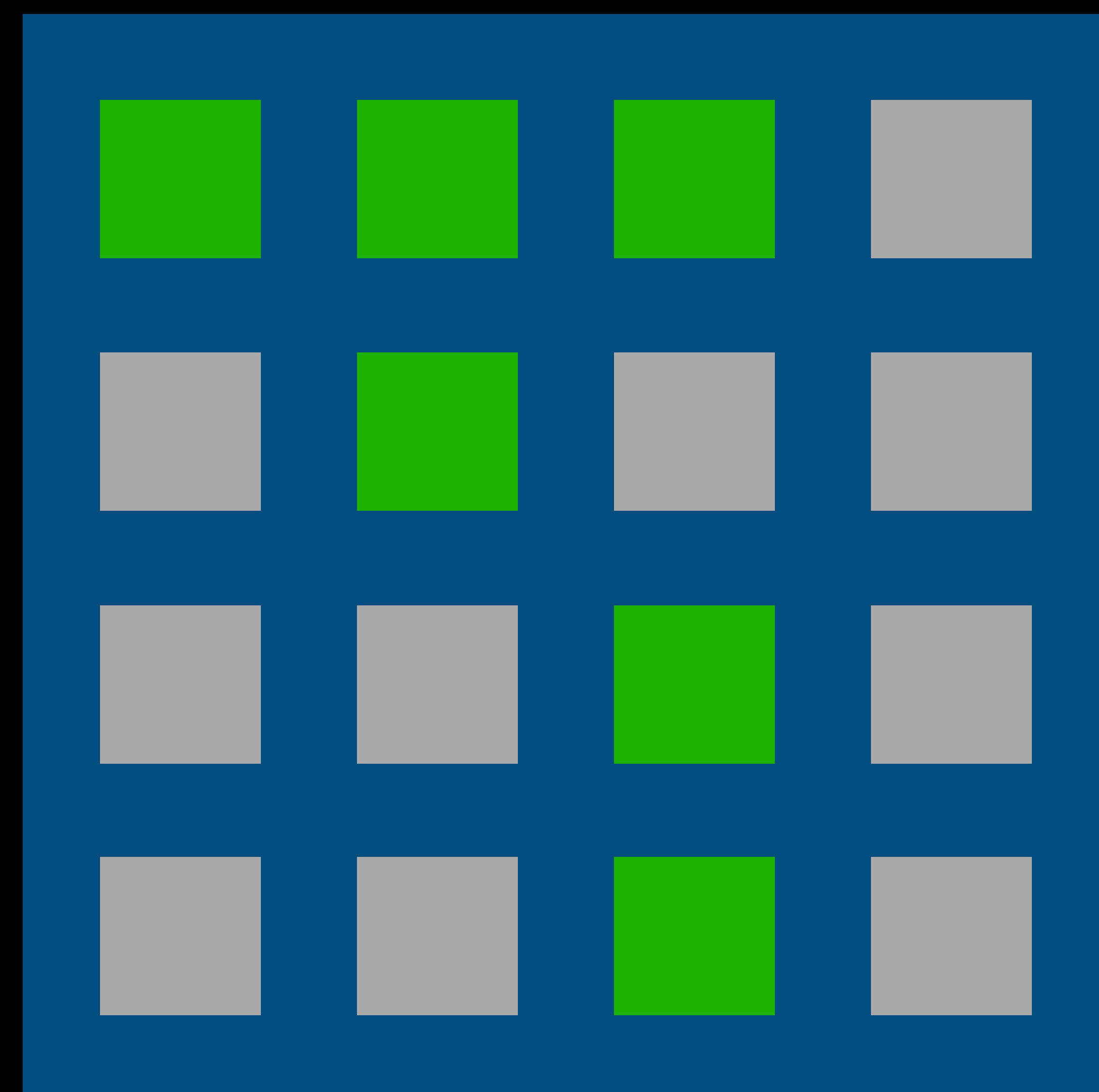



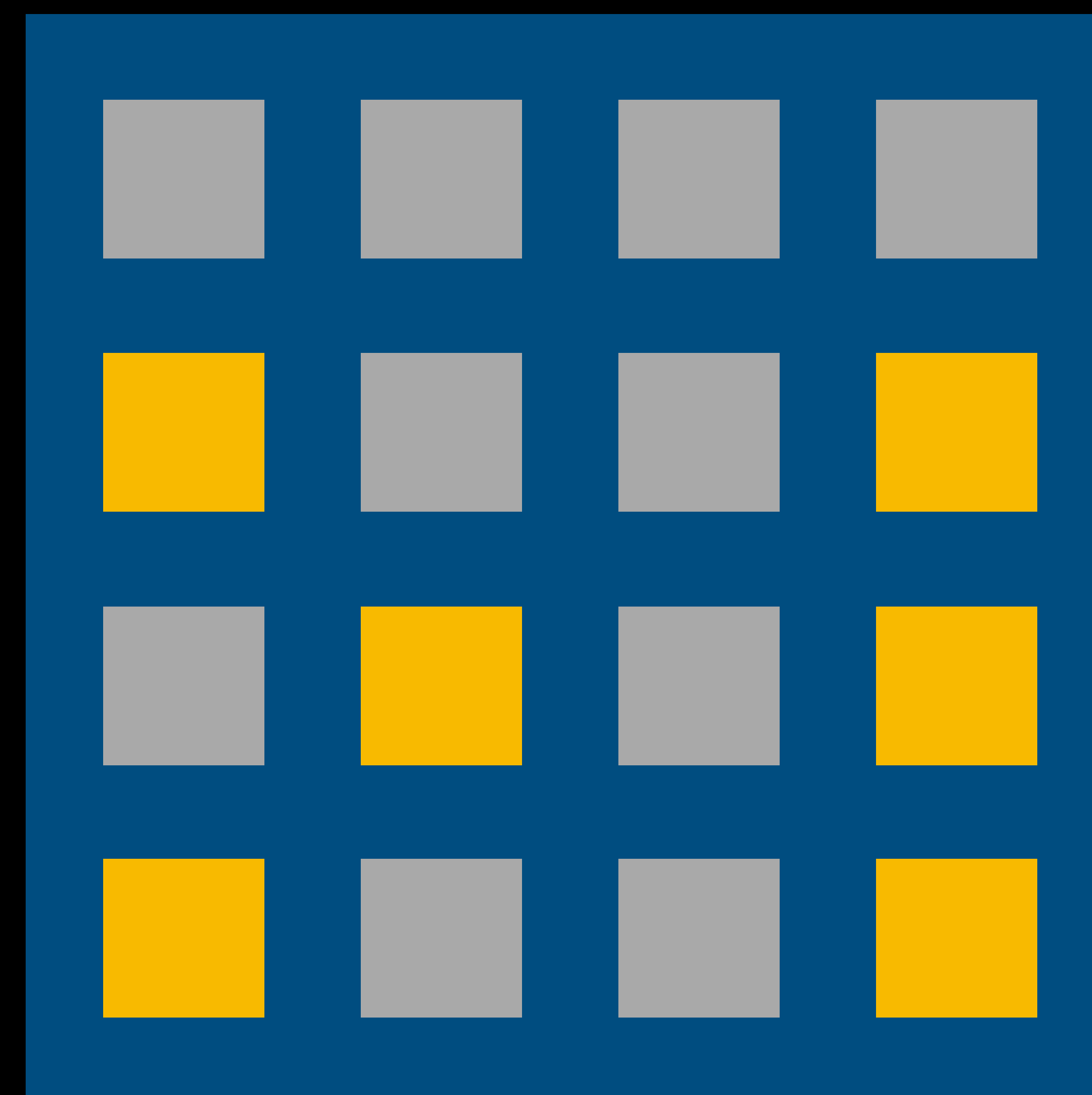
Meshing

Update page tables with `mmap`, `memfd`, `MAP_SHARED`

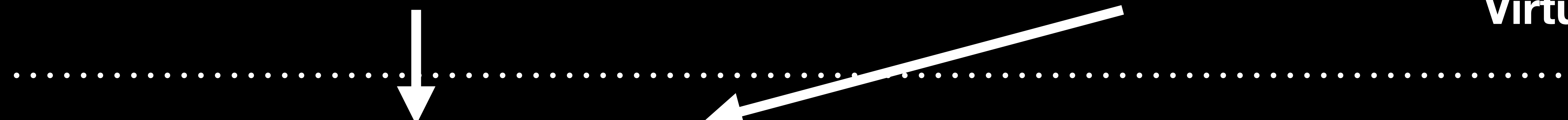
Meshing



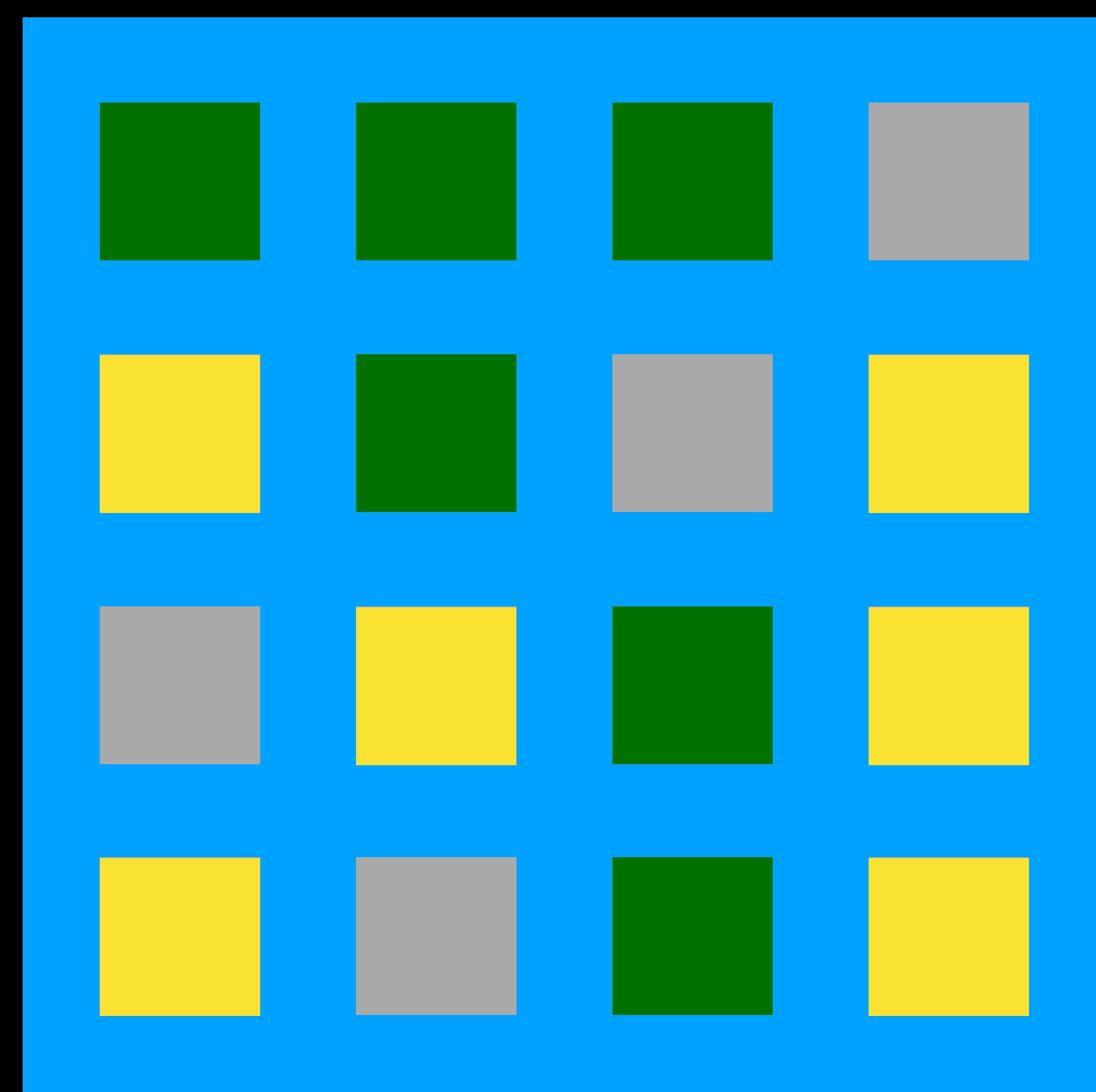


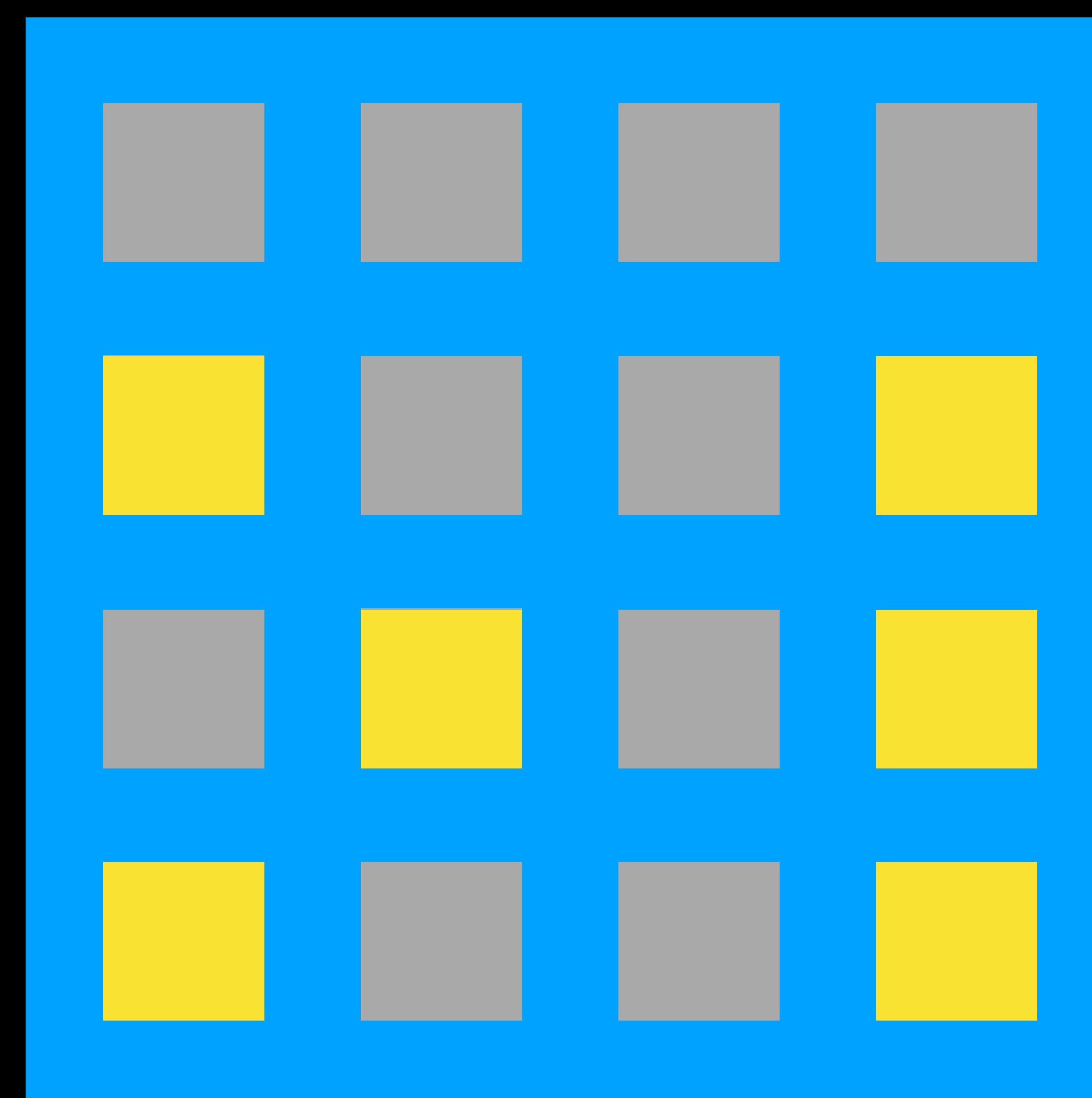
Mark virtual page read/write

Virtual

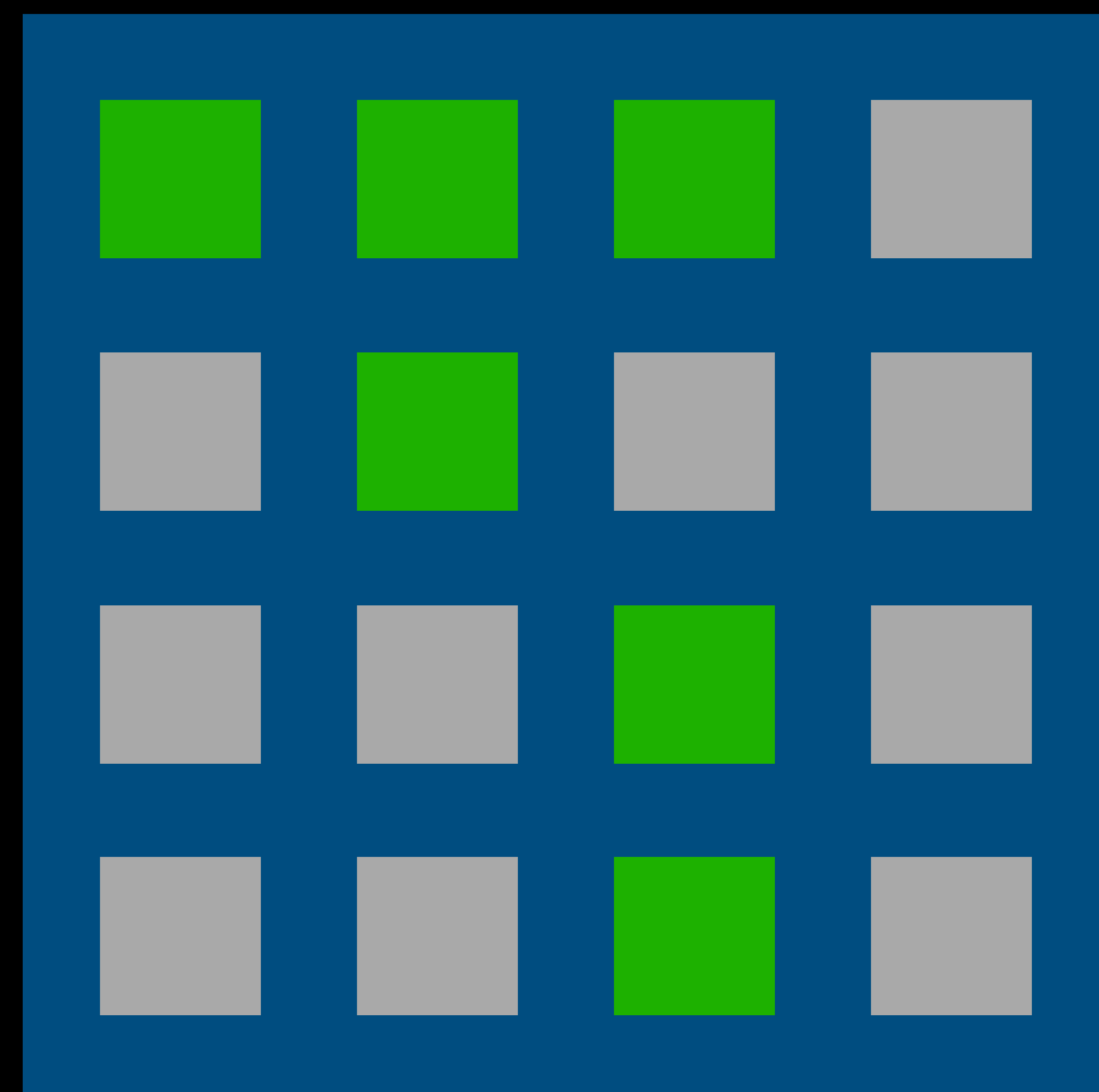
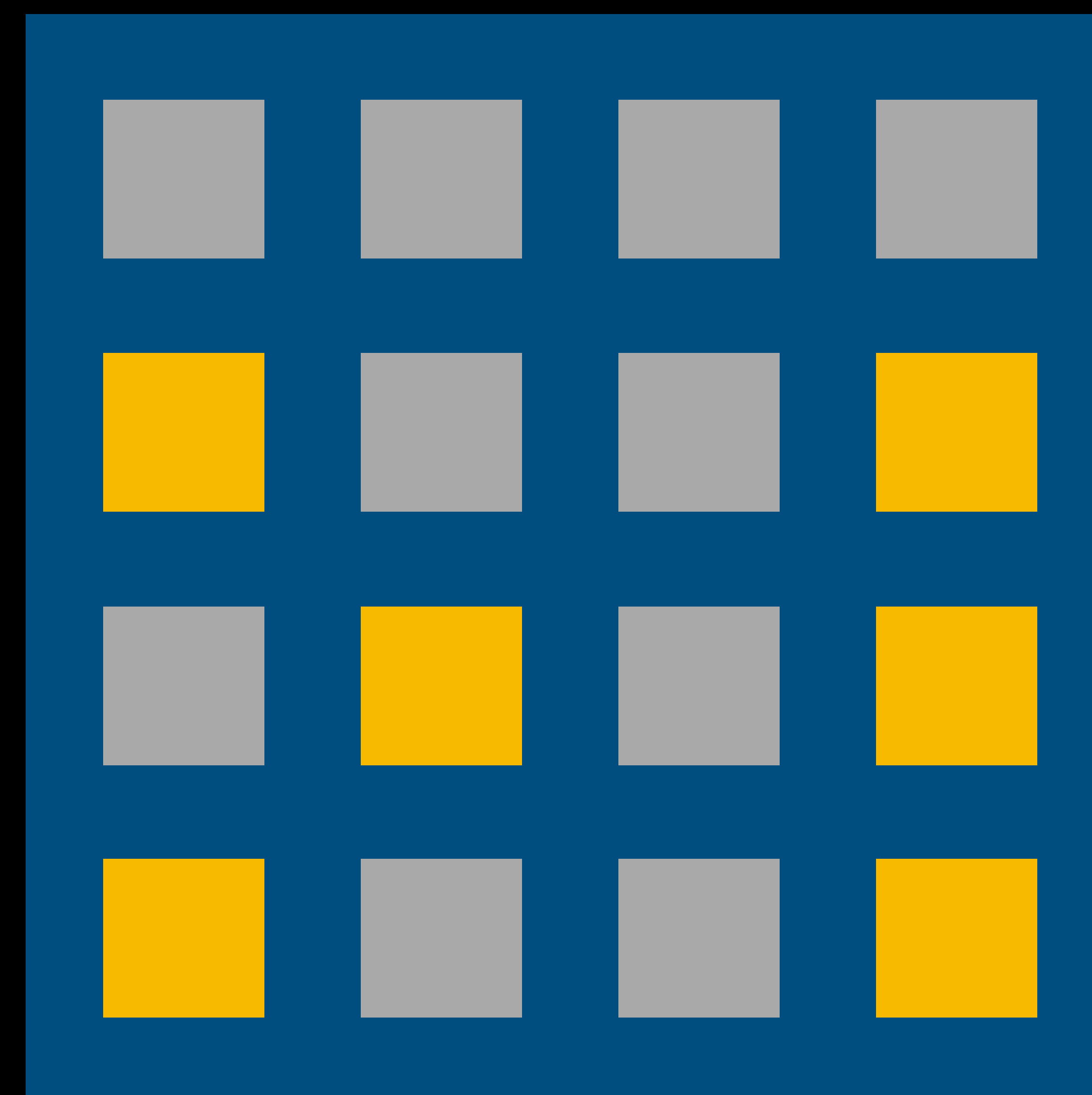
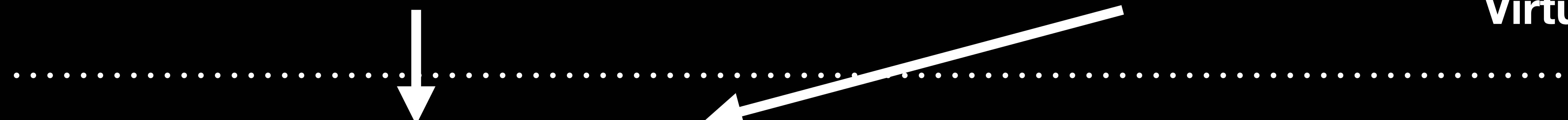


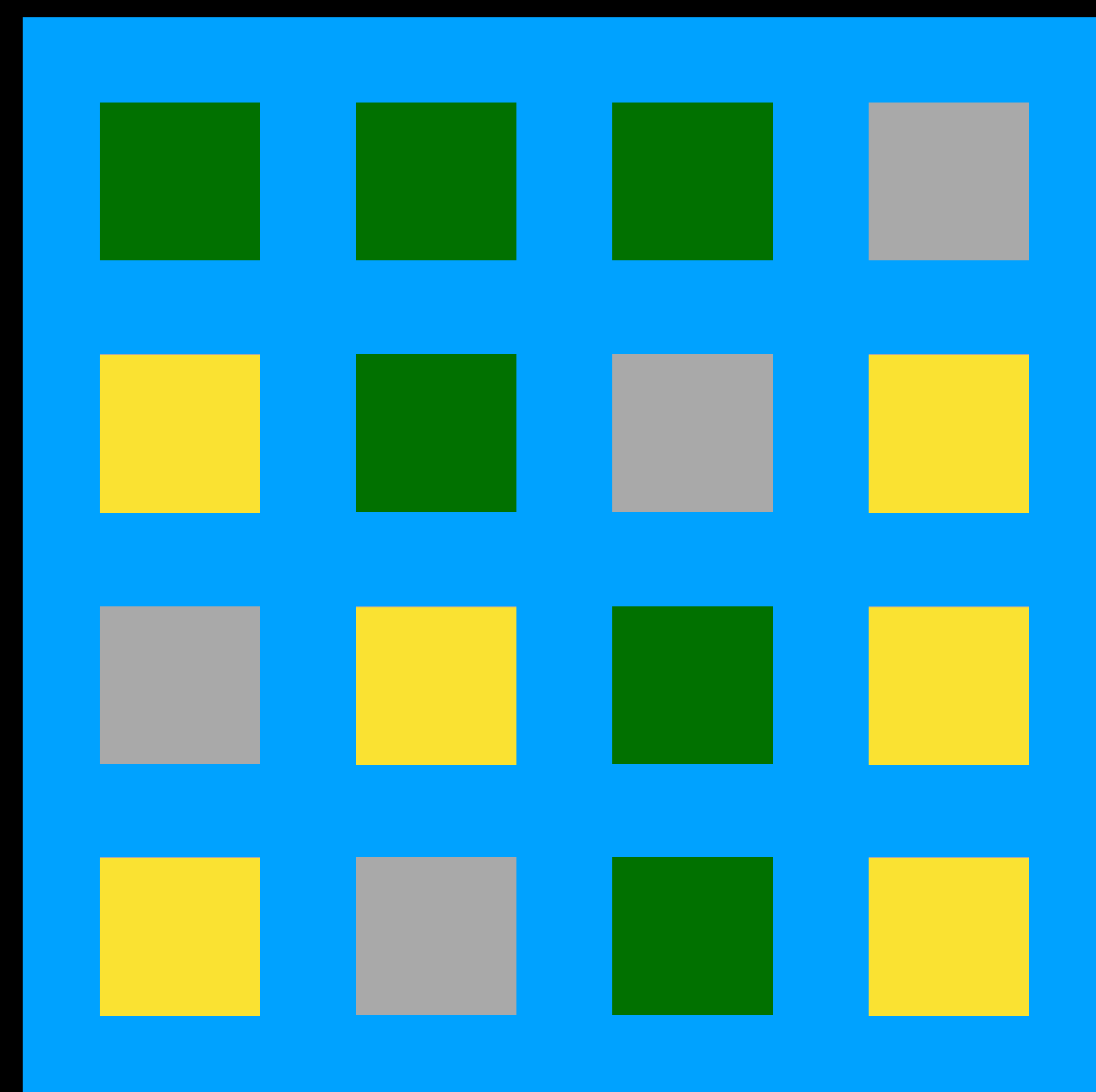
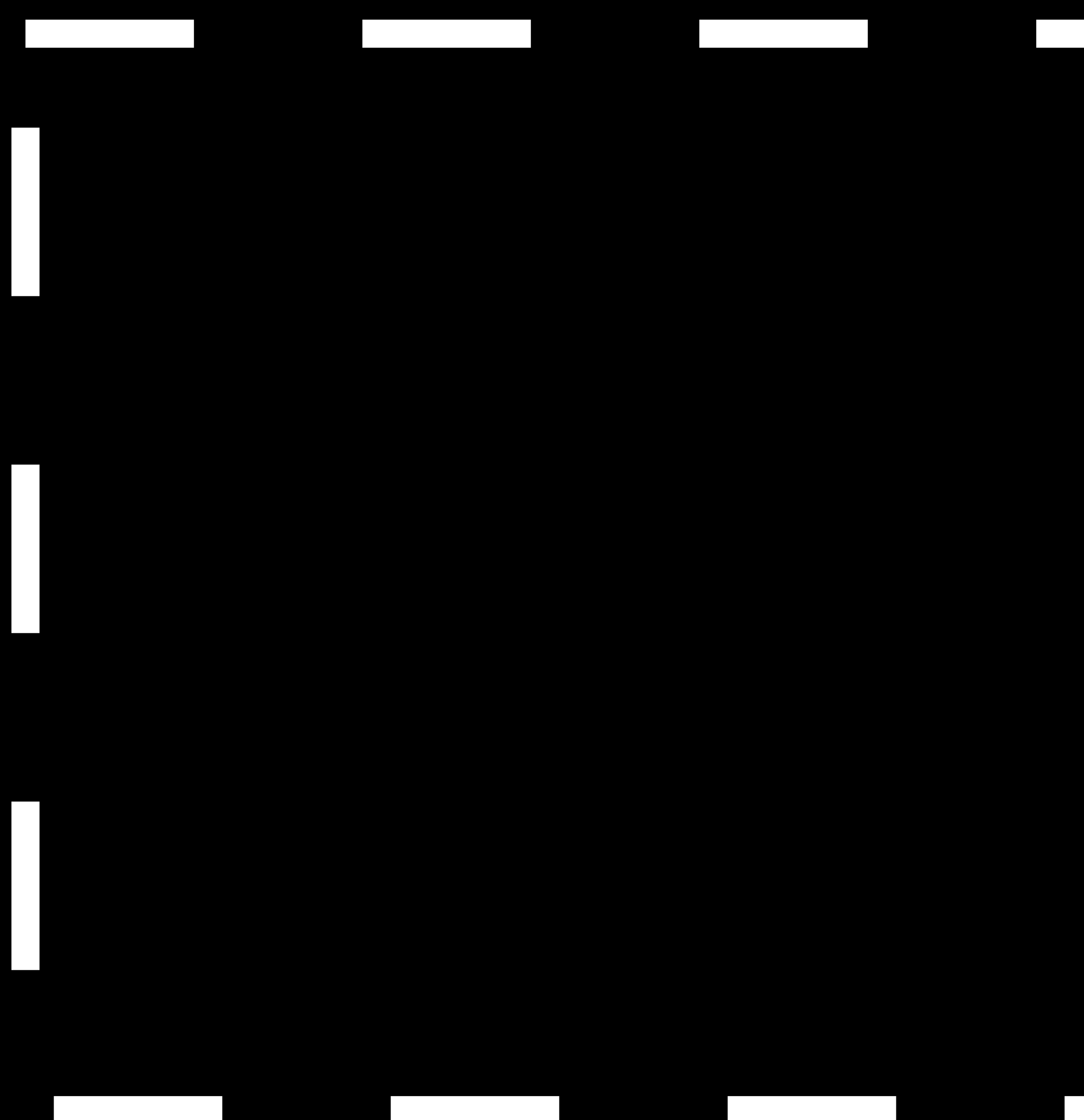
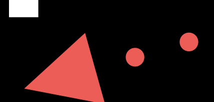
Physical



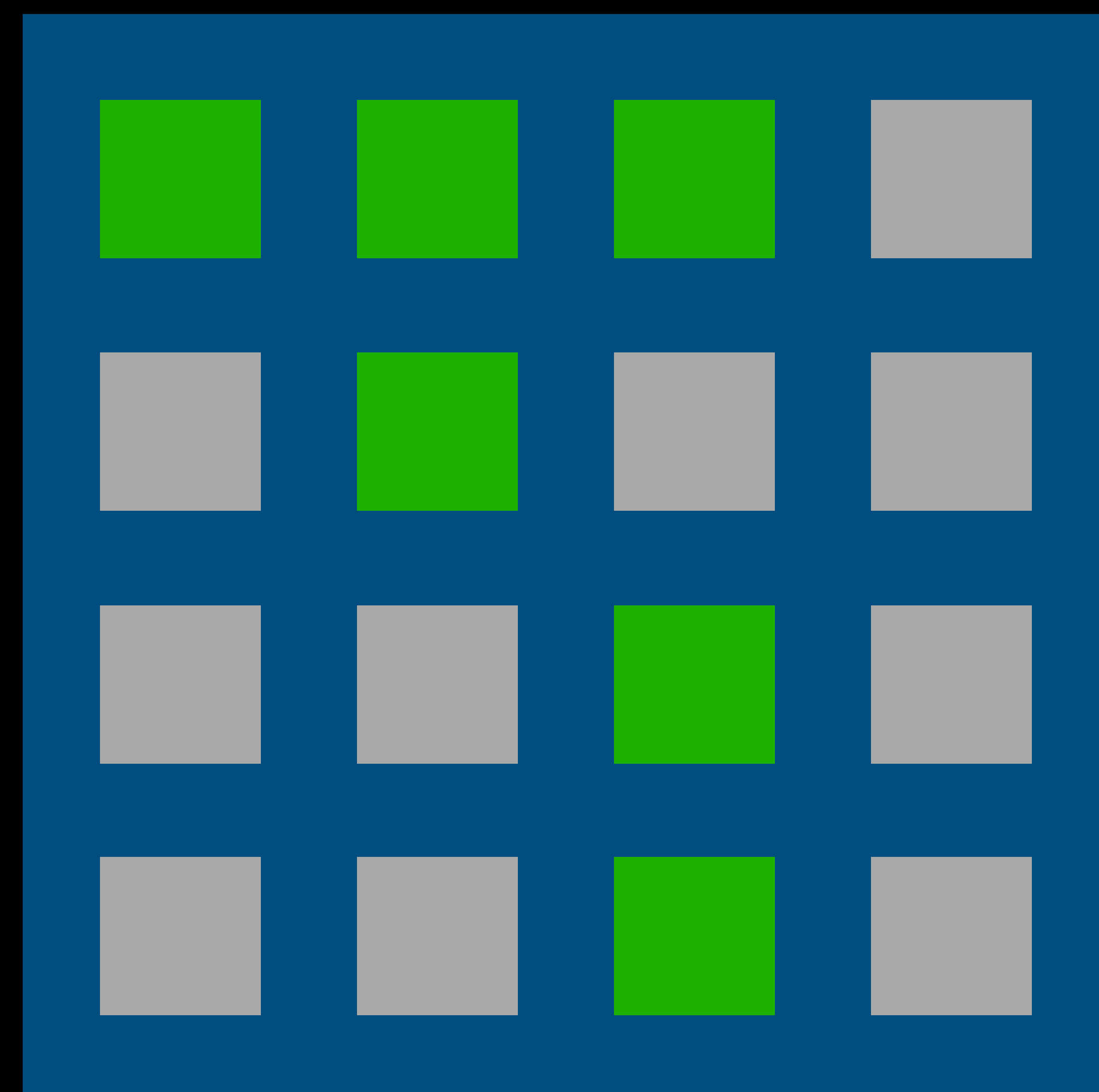


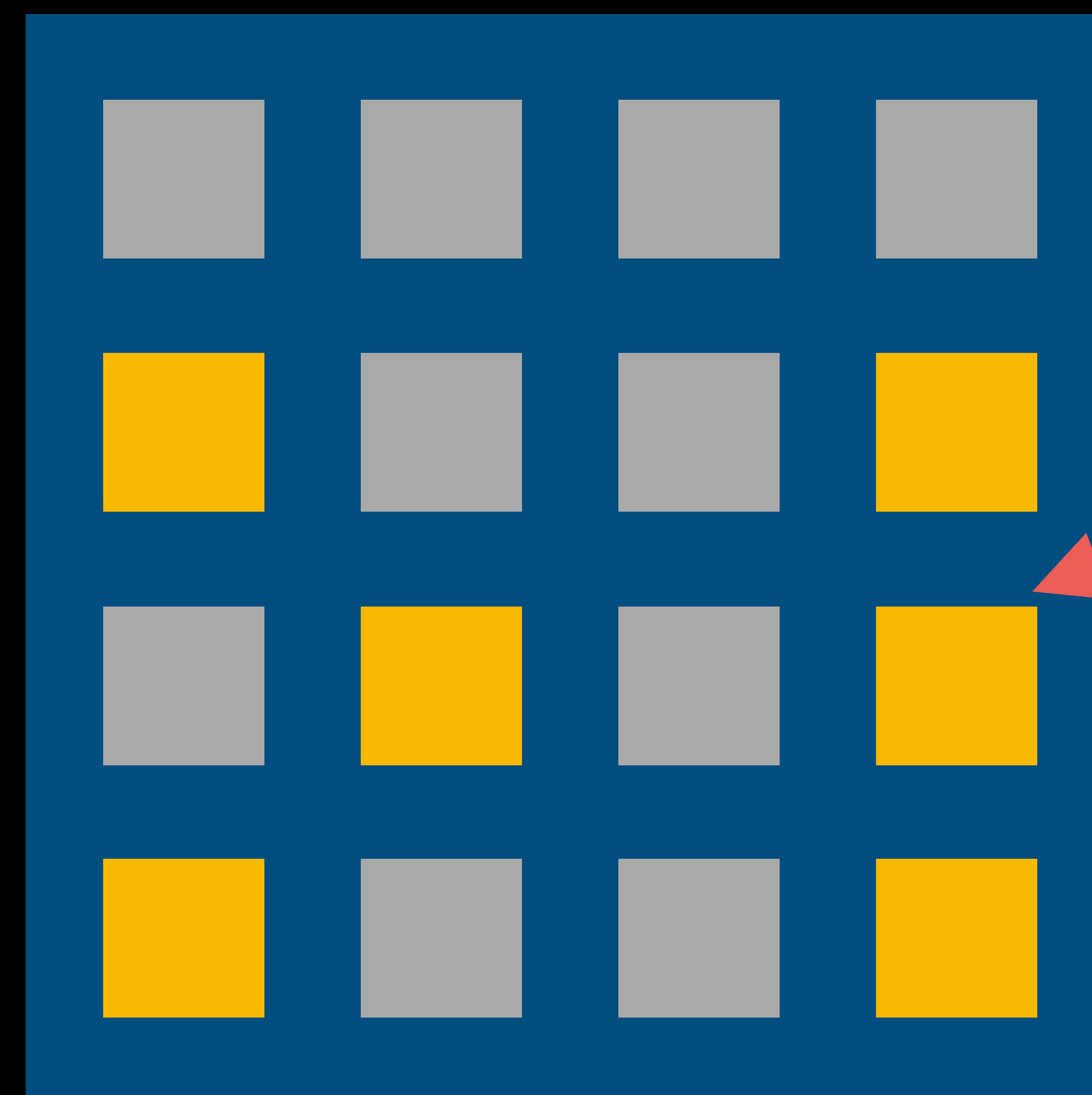
Meshing

Meshing

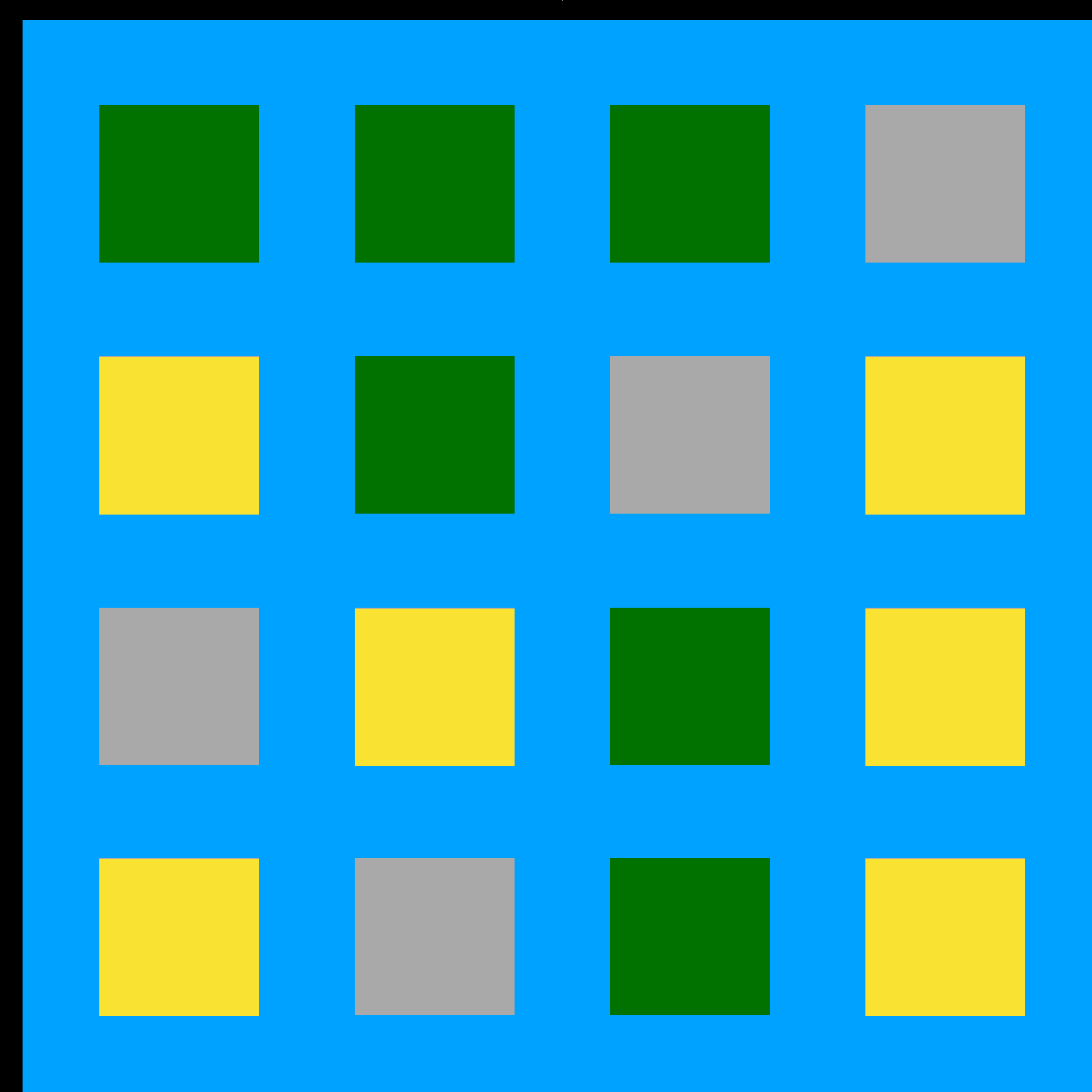


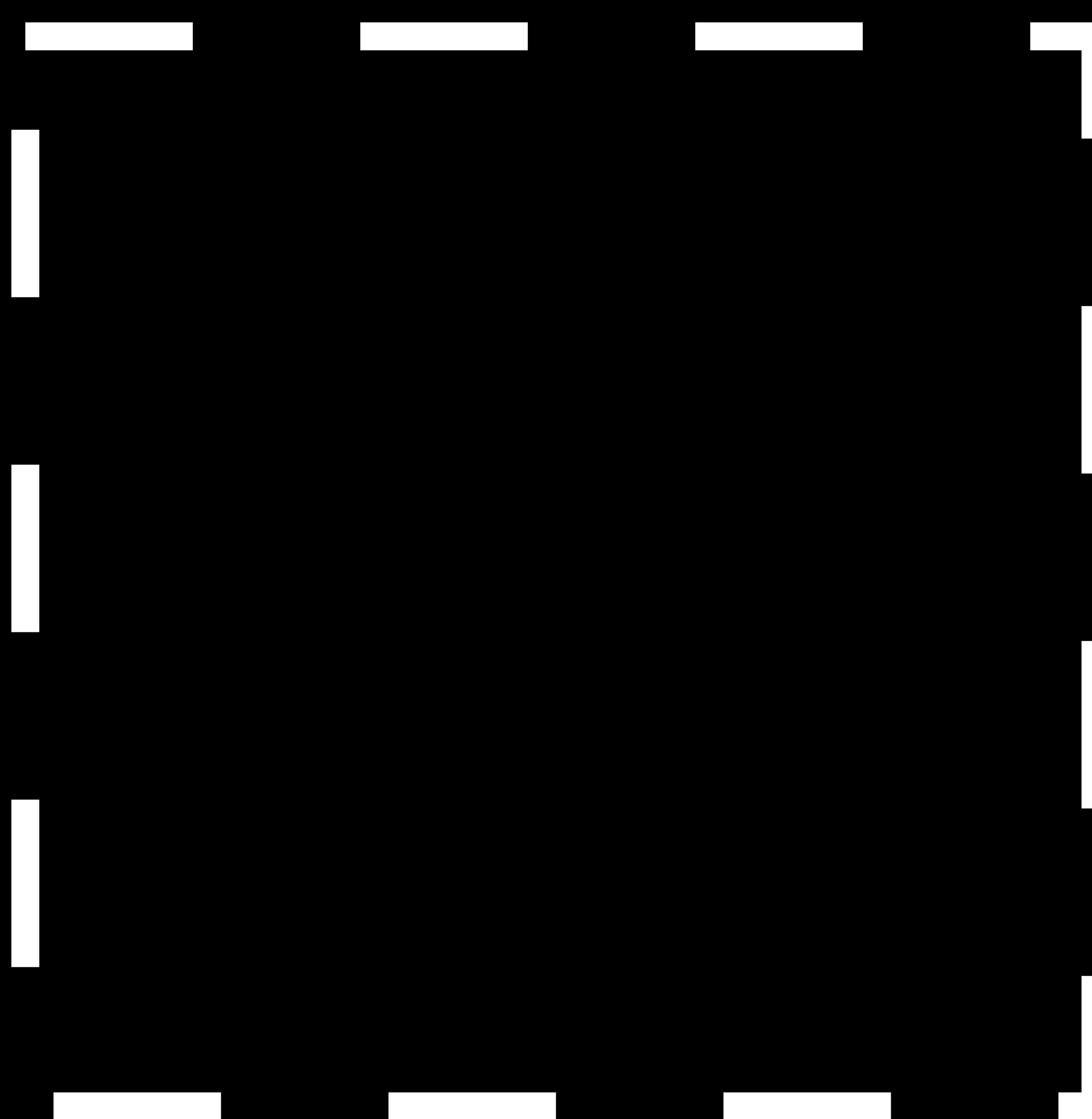


No virtual addresses changed!

Virtual

Physical

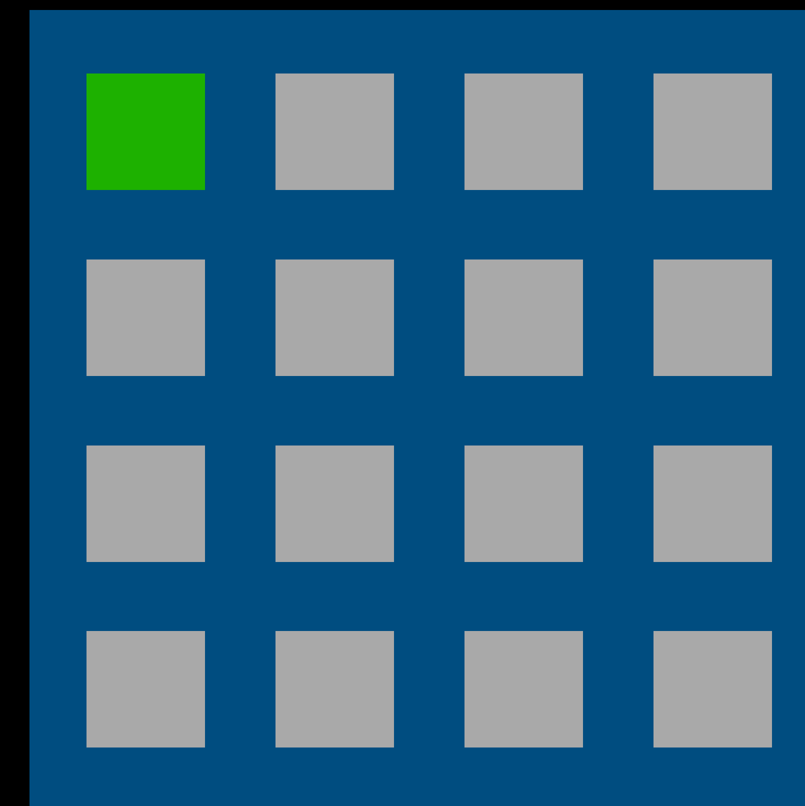


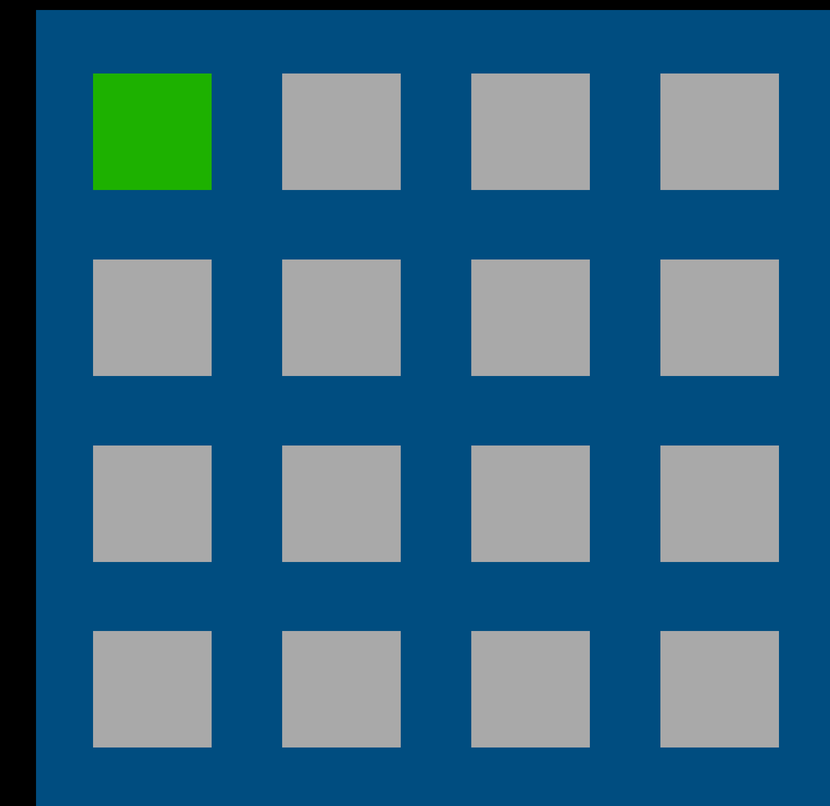


Worst Case:

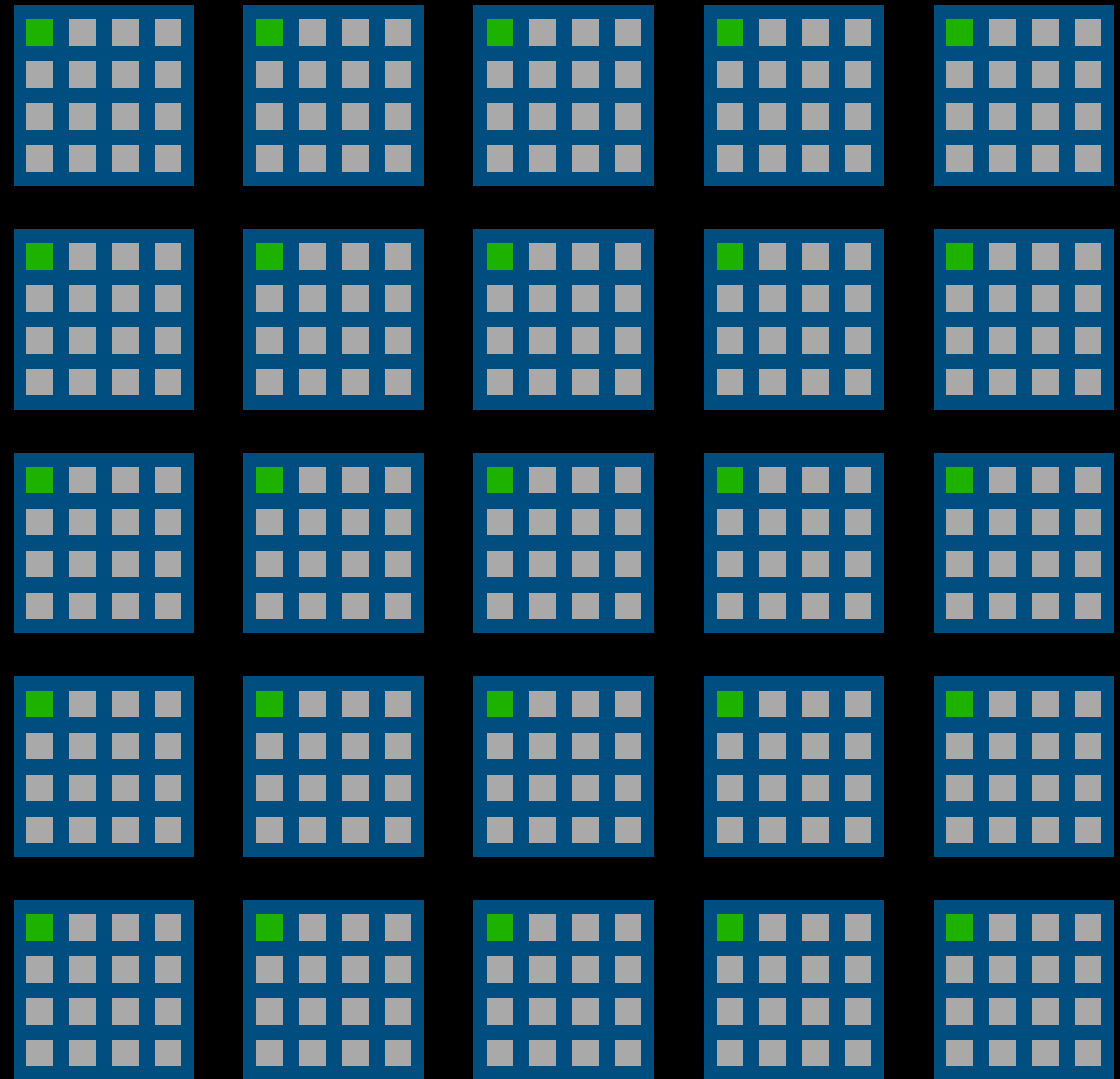
Worst Case:

low occupancy, non-meshable pages



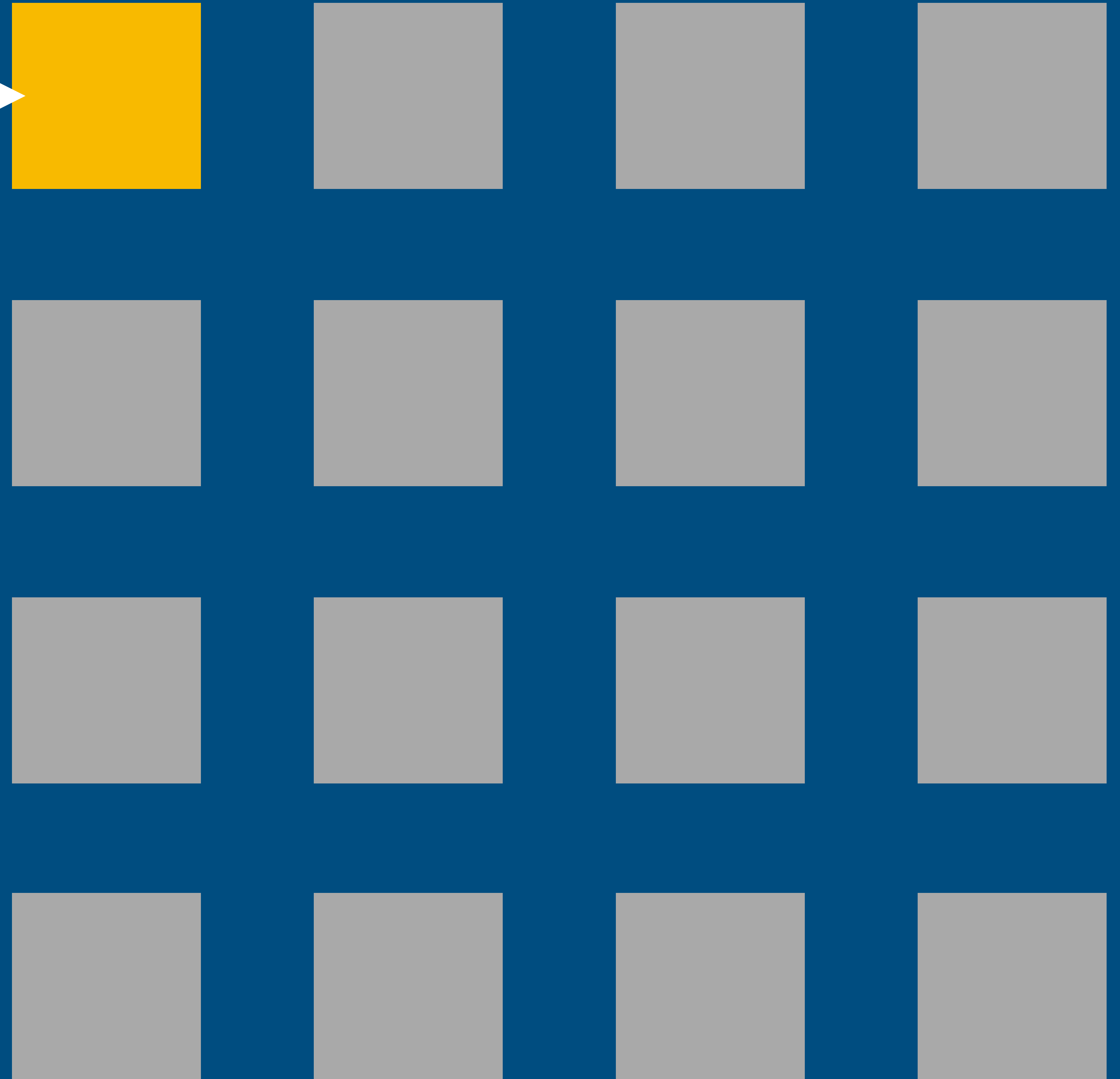


Worst Case:
many
low occupancy,
non-meshable
pages



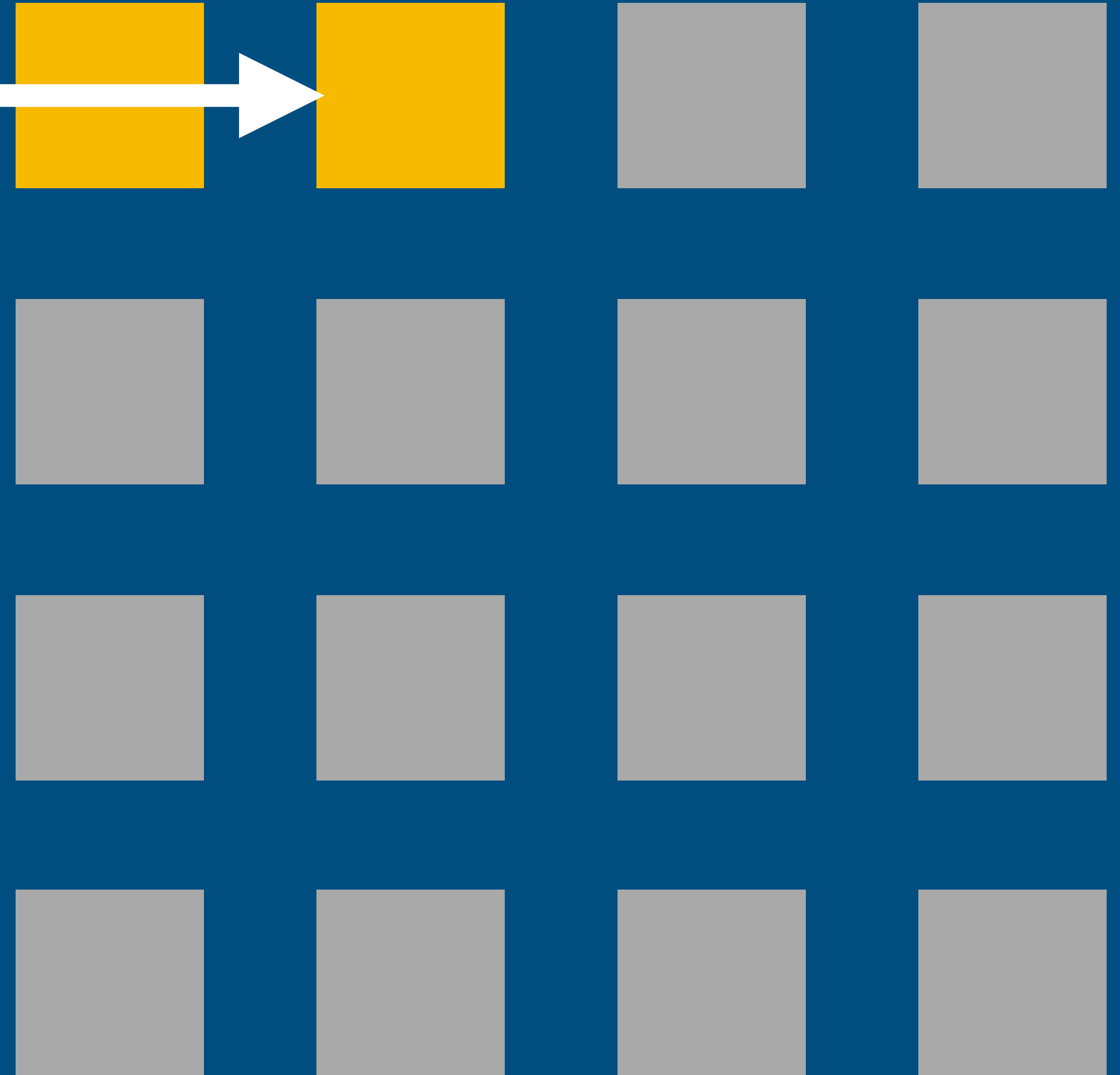
Standard allocators

```
malloc(256)
```



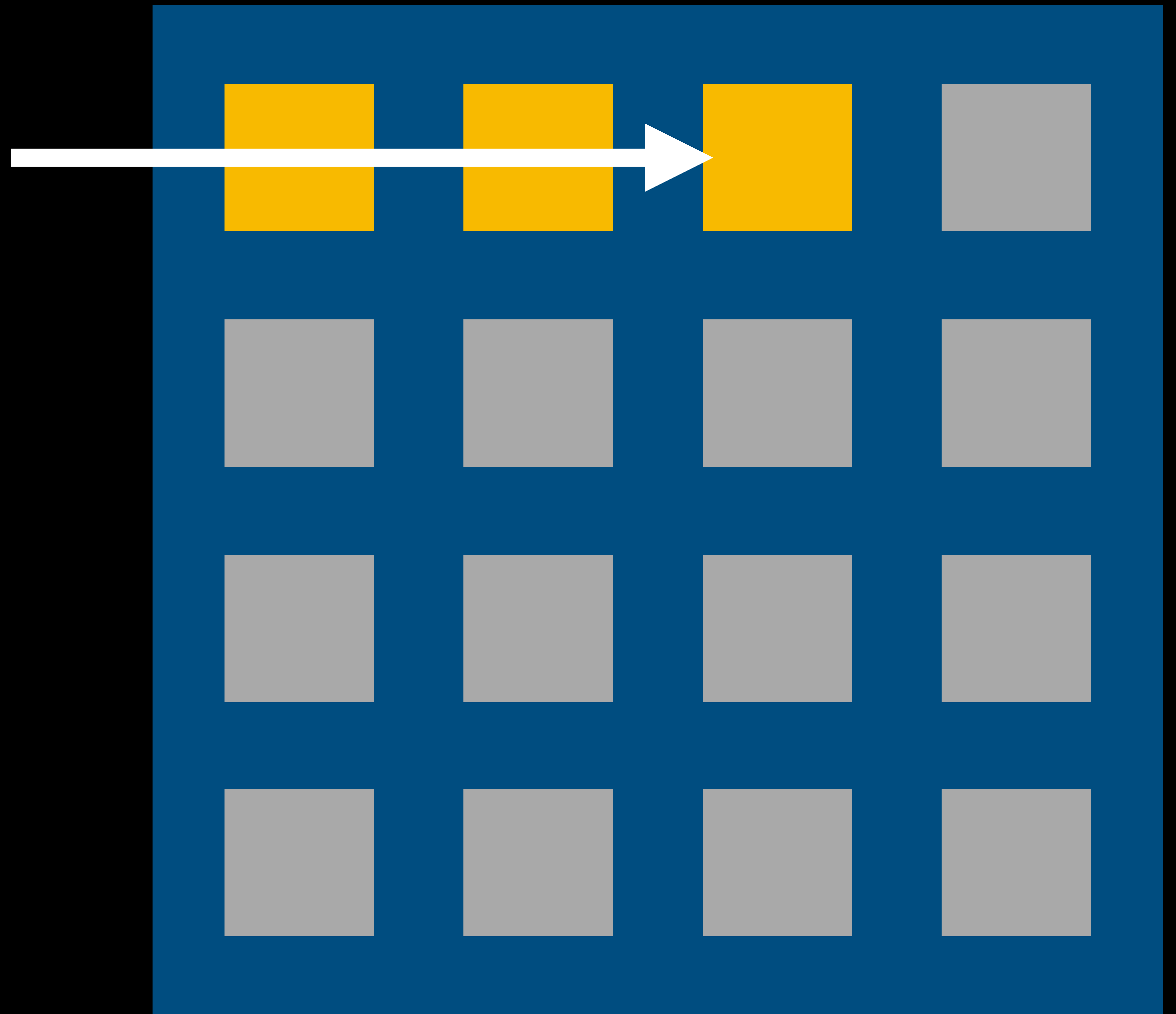
Standard allocators

`malloc(256)`



Standard allocators

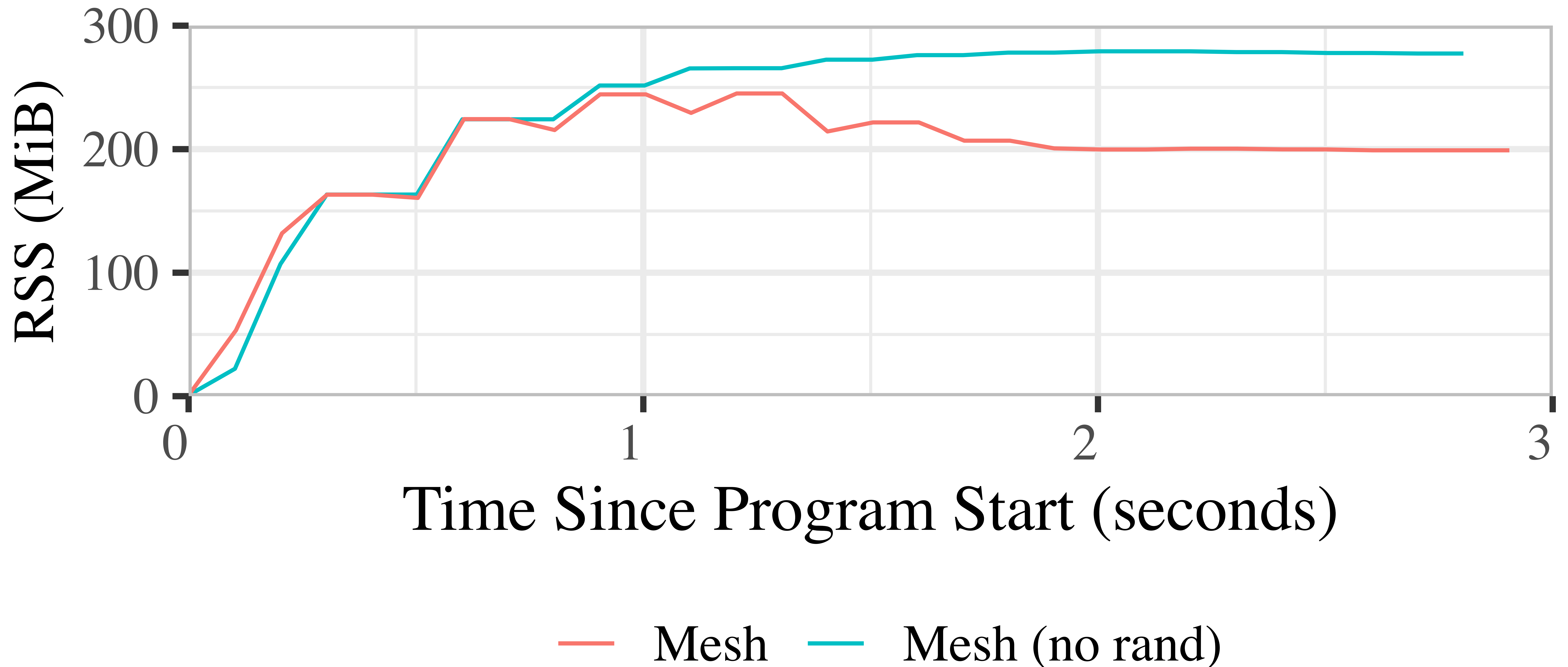
```
malloc(256)
```



Standard allocators

Mesh uses **randomization** to ensure live objects are uniformly distributed

Regular allocation patterns are real

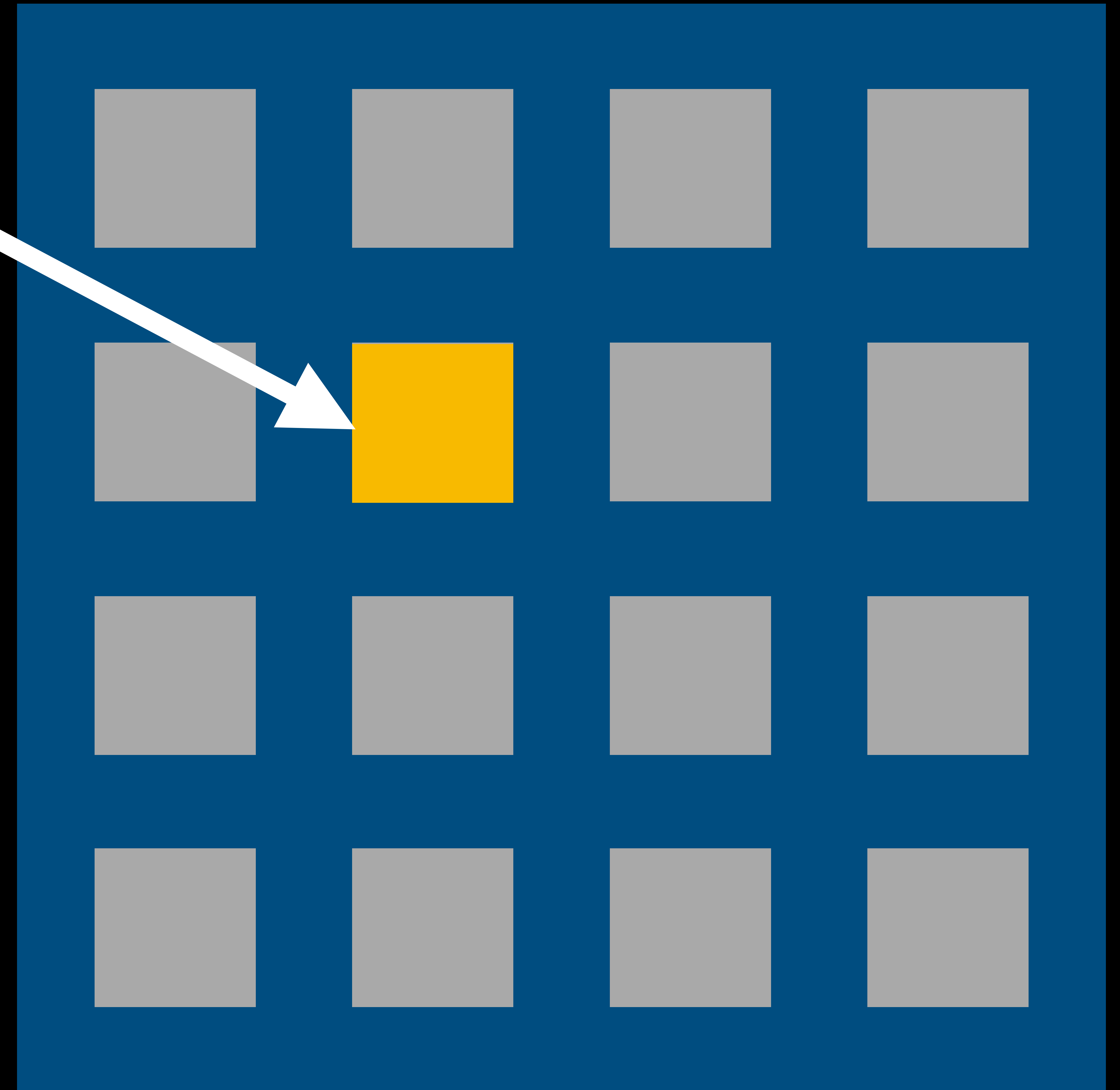


How to **randomize** allocation?

Random probing:

```
while true:
  if rand_off().is_free:
    return rand_off
```

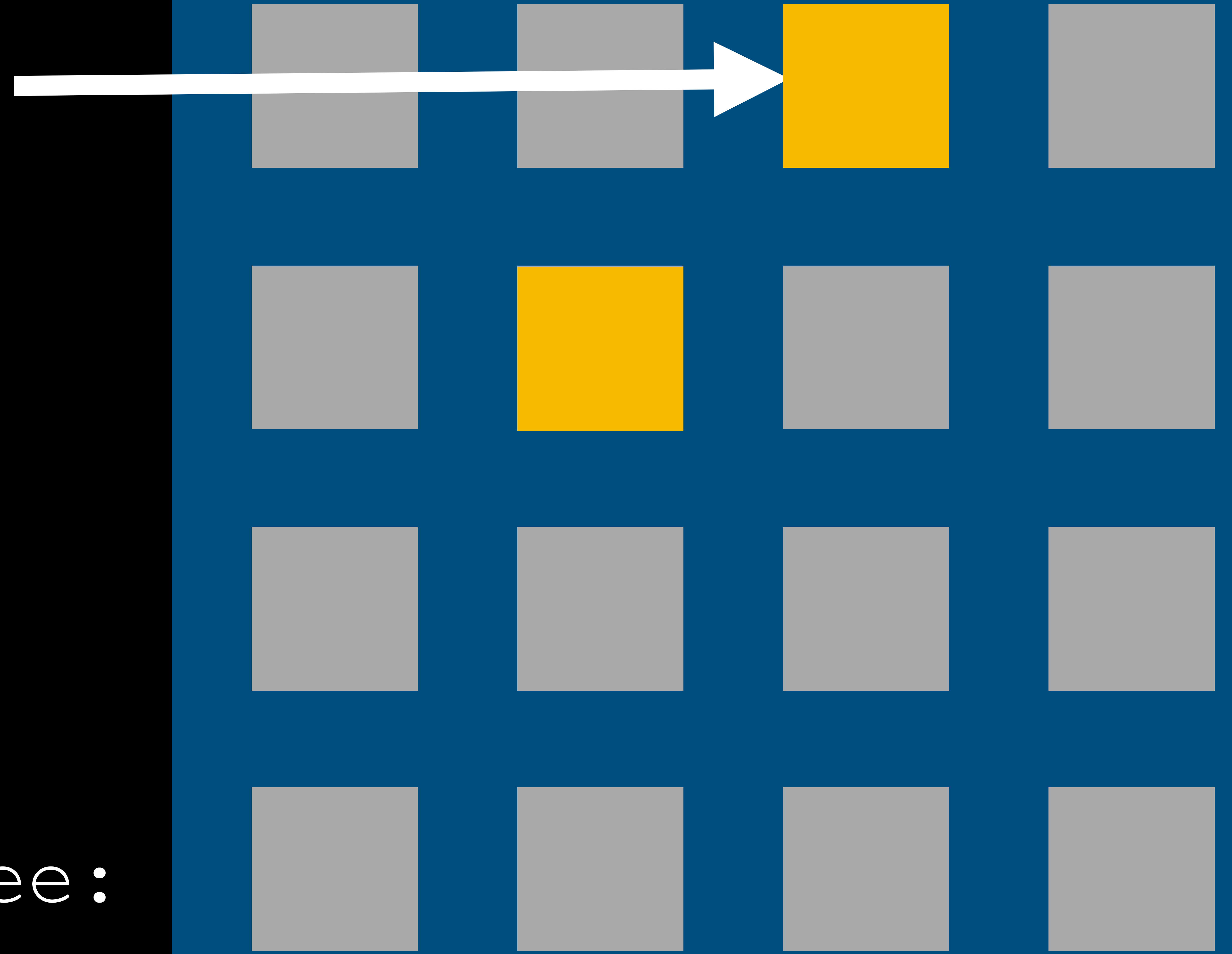
```
malloc(256)
```



Random probing:

```
while true:
  if rand_off().is_free:
    return rand_off
```

```
malloc(256)
```



Random probing:

```
while true:
  if rand_off().is_free:
    return rand_off
```

```
malloc(256)
```

Random probing:

```
while true:
  if rand_off().is_free:
    return rand_off
```

```
malloc(256)
```

Random probing:

```
while true:
  if rand_off().is_free:
    return rand_off
```

(DieHard [Berger & Zorn 2006])

Random probing fast in expectation *iff page occupancy is low*

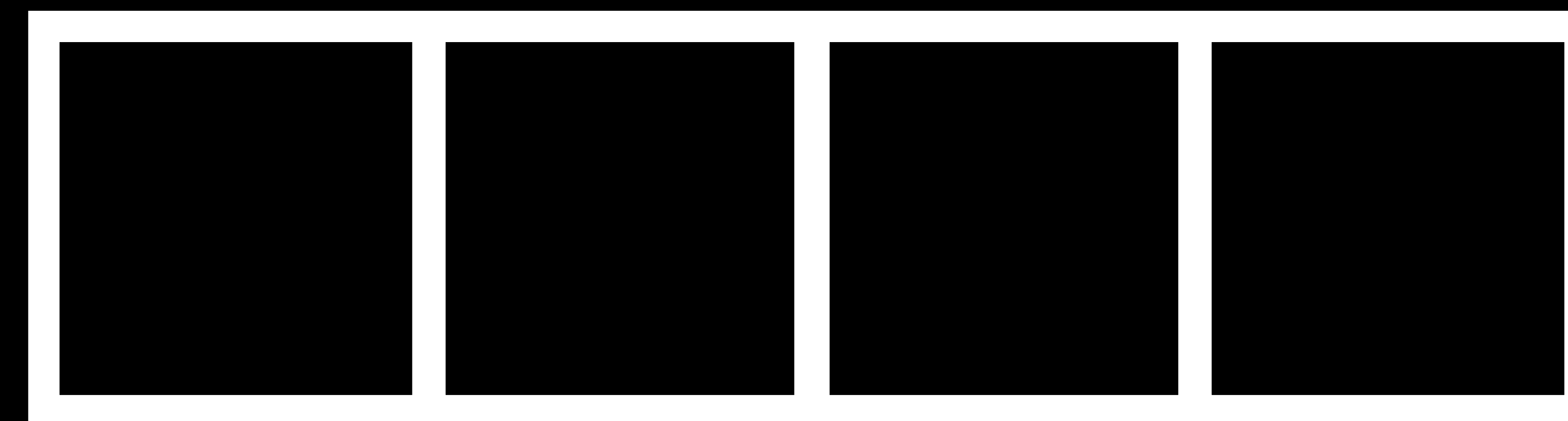
Random probing fast in expectation *iff page occupancy is low*

but this is at odds with minimizing heap size!

Shuffle Vector: Fast randomized allocation + full page utilization

Shuffle Vector: Fast randomized allocation

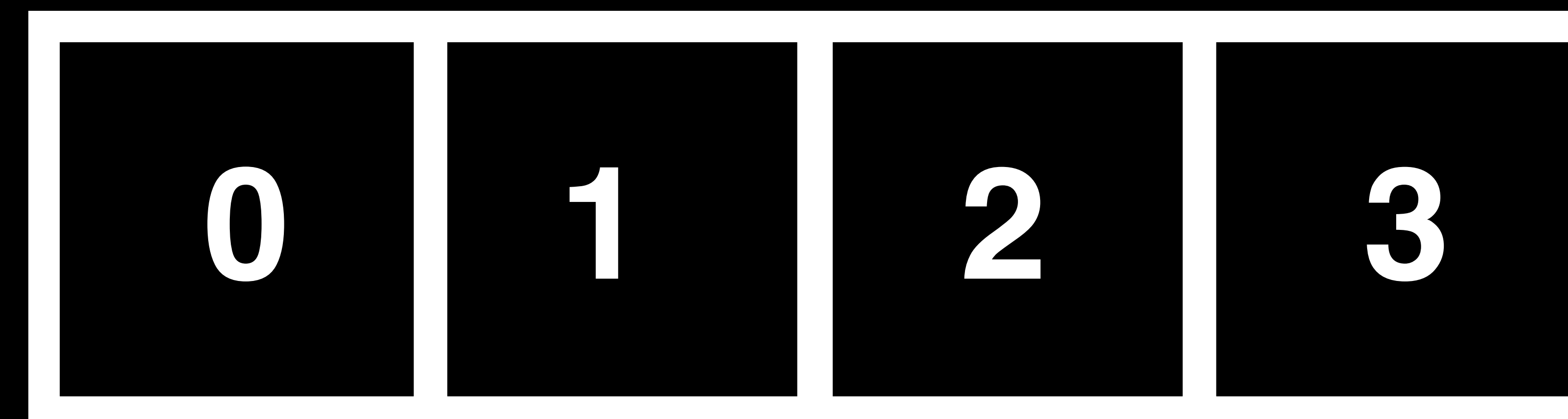
```
load
```



Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

`load`



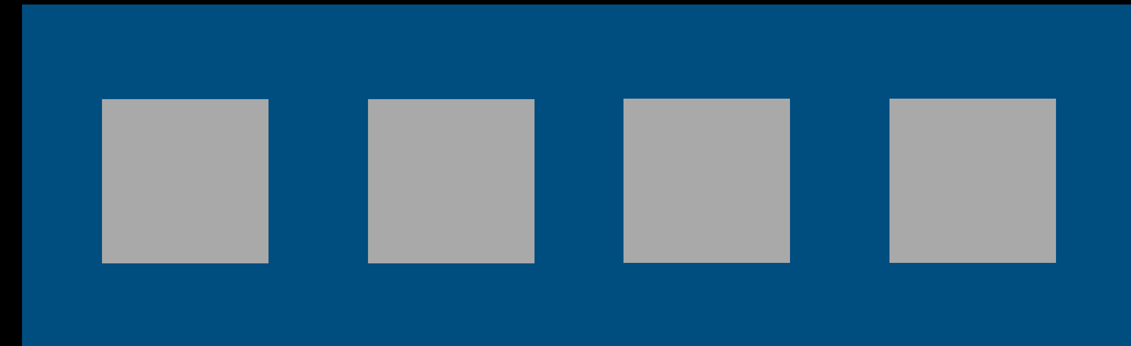
Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
shuffle( [0][1][2][3] )
```

Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation



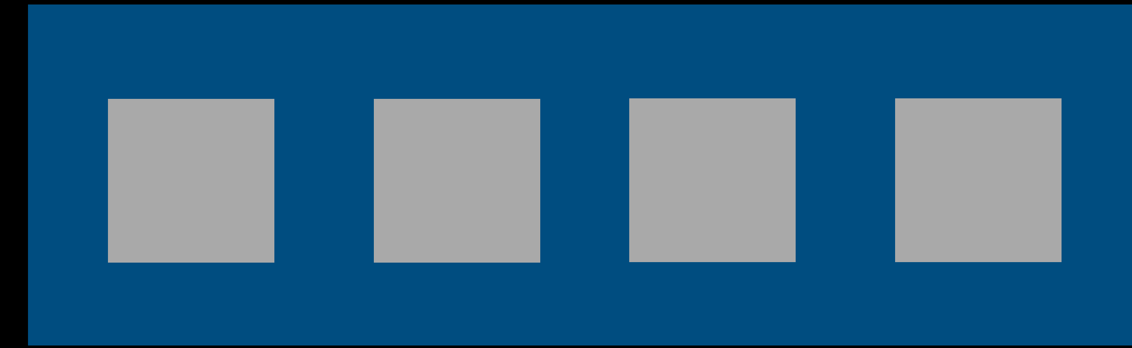
shuffle(| 2 | 3 | 1 | 0 |)

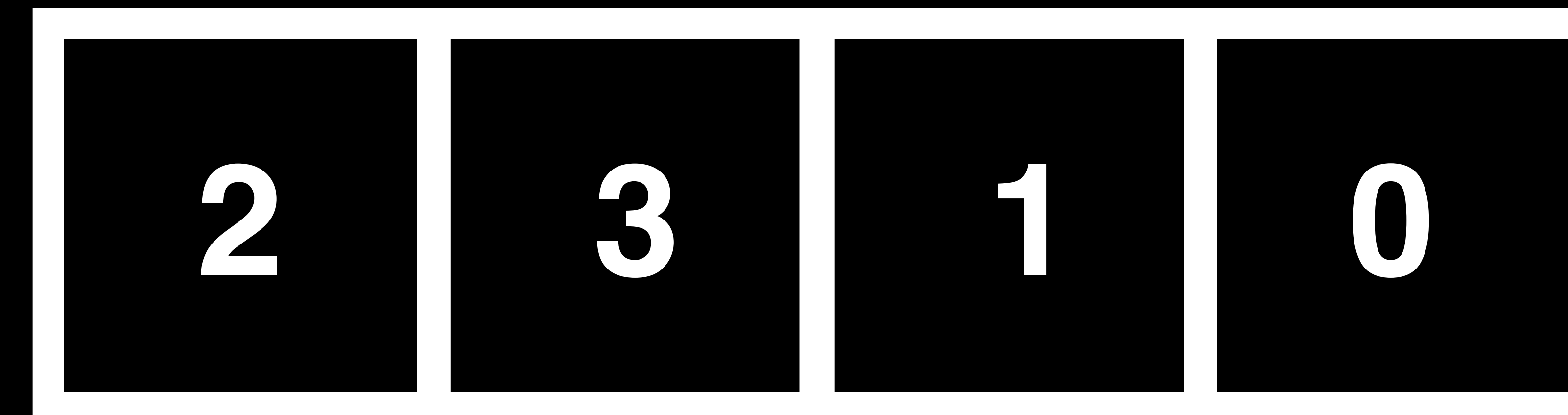
Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
malloc()
```

Page

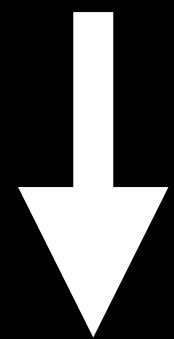




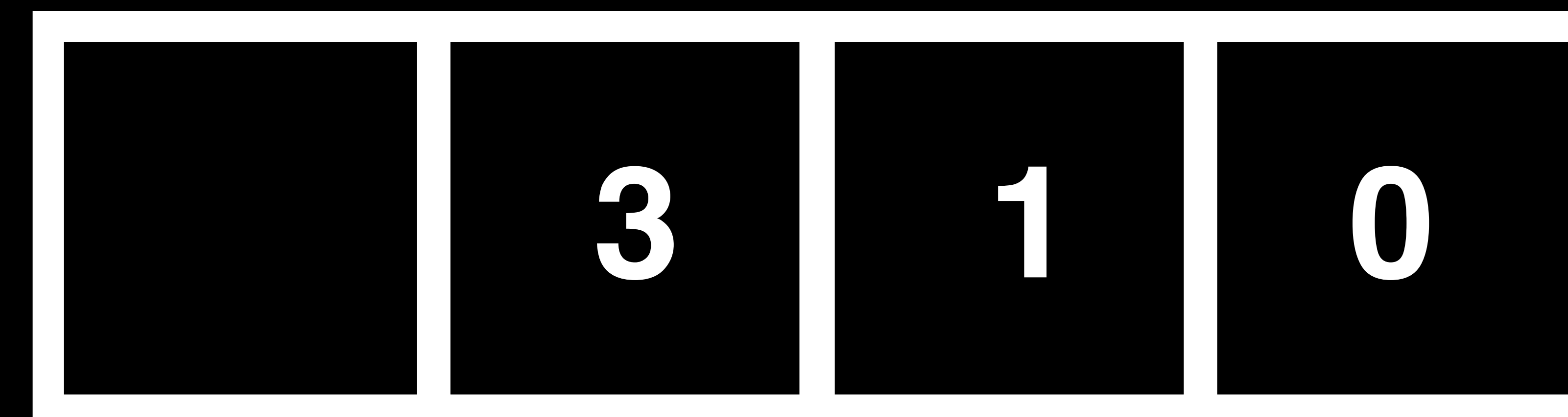
Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
malloc()
```



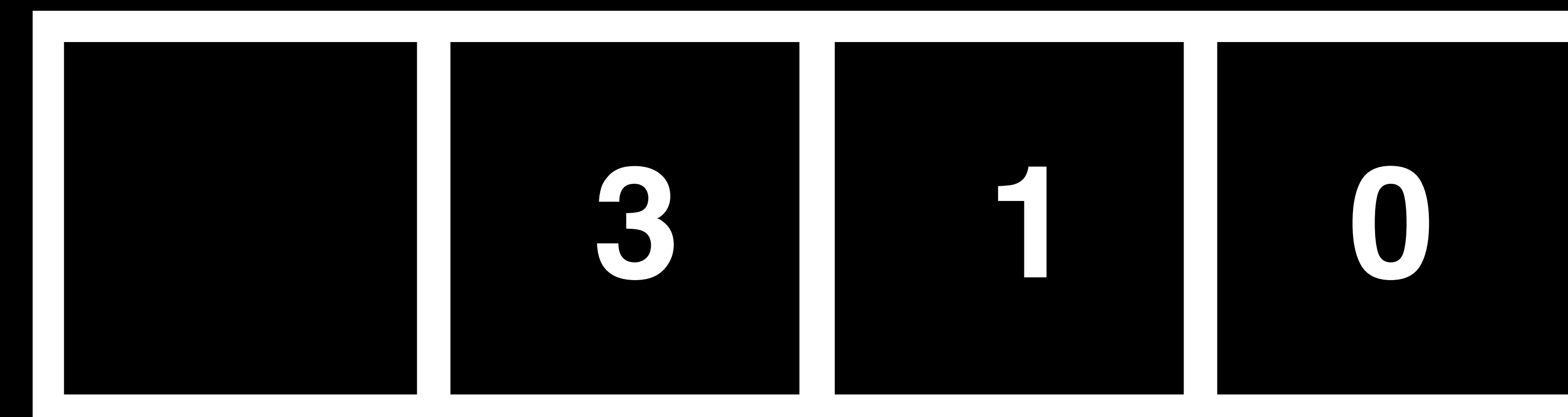
```
page_start +
2 * object_size
```



Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
free(2)
```

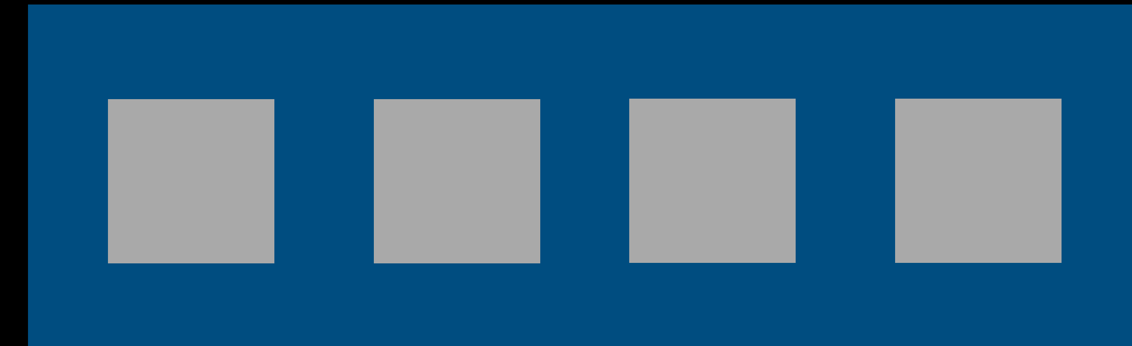


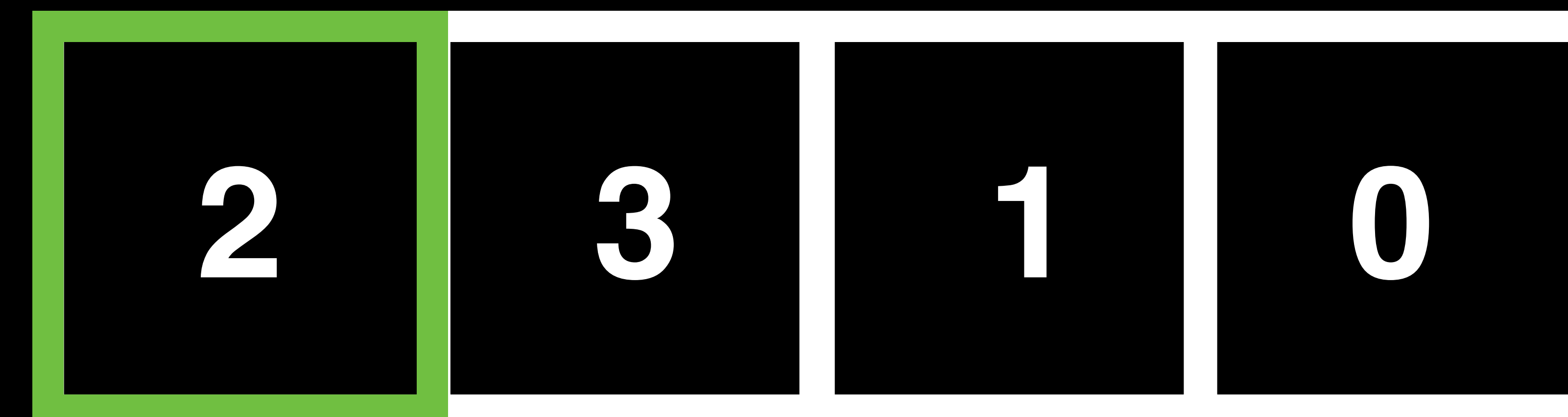
Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
free( )
```

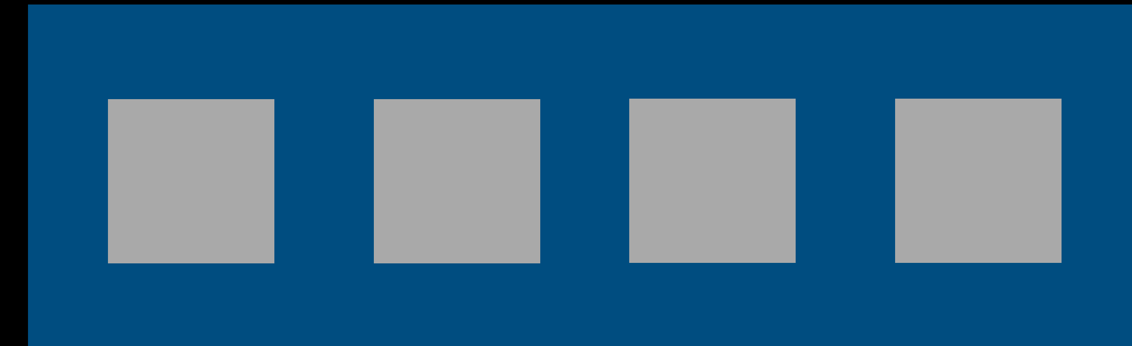
Page





Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation




```
free( )
```

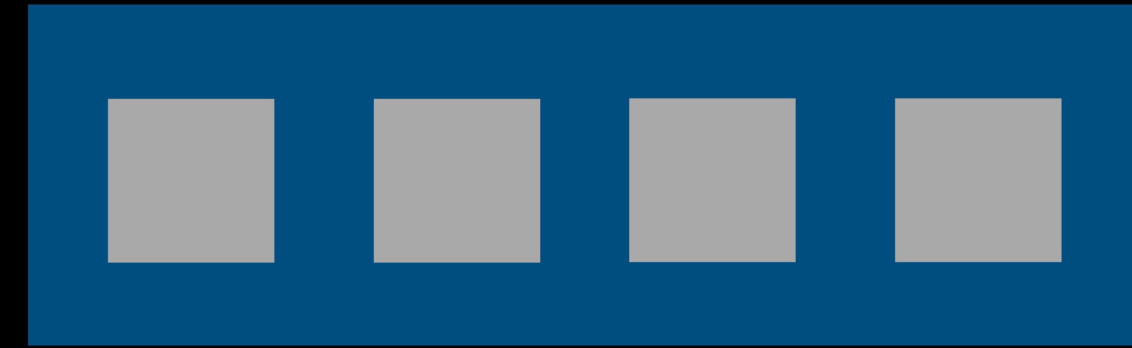
```
shuffle_one(
```

2	3	1	0

```
)
```

Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation



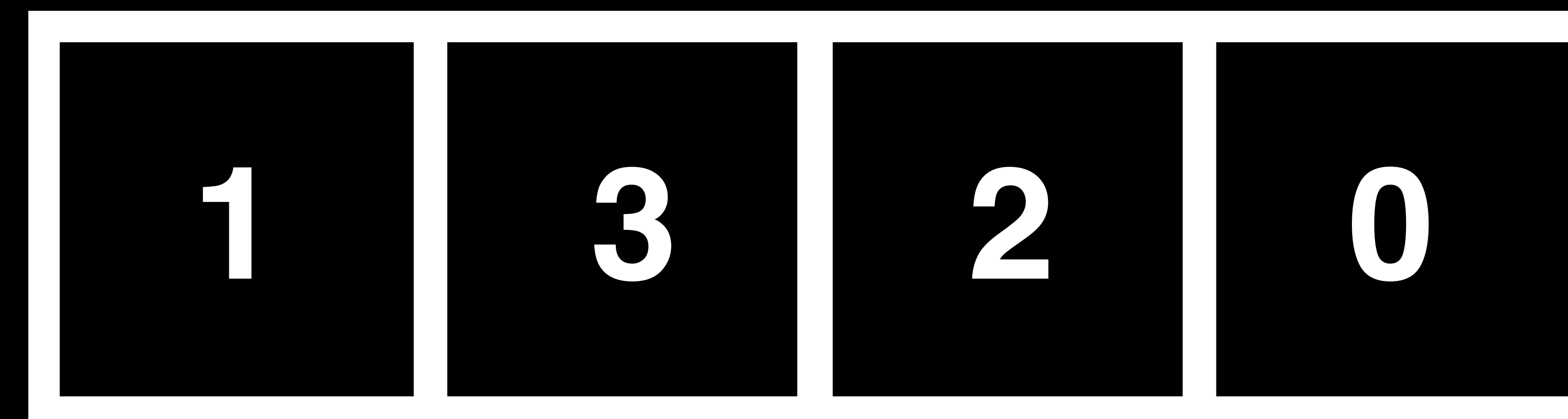
`free( )`

`shuffle_one(` 1 3 2 0 `)`

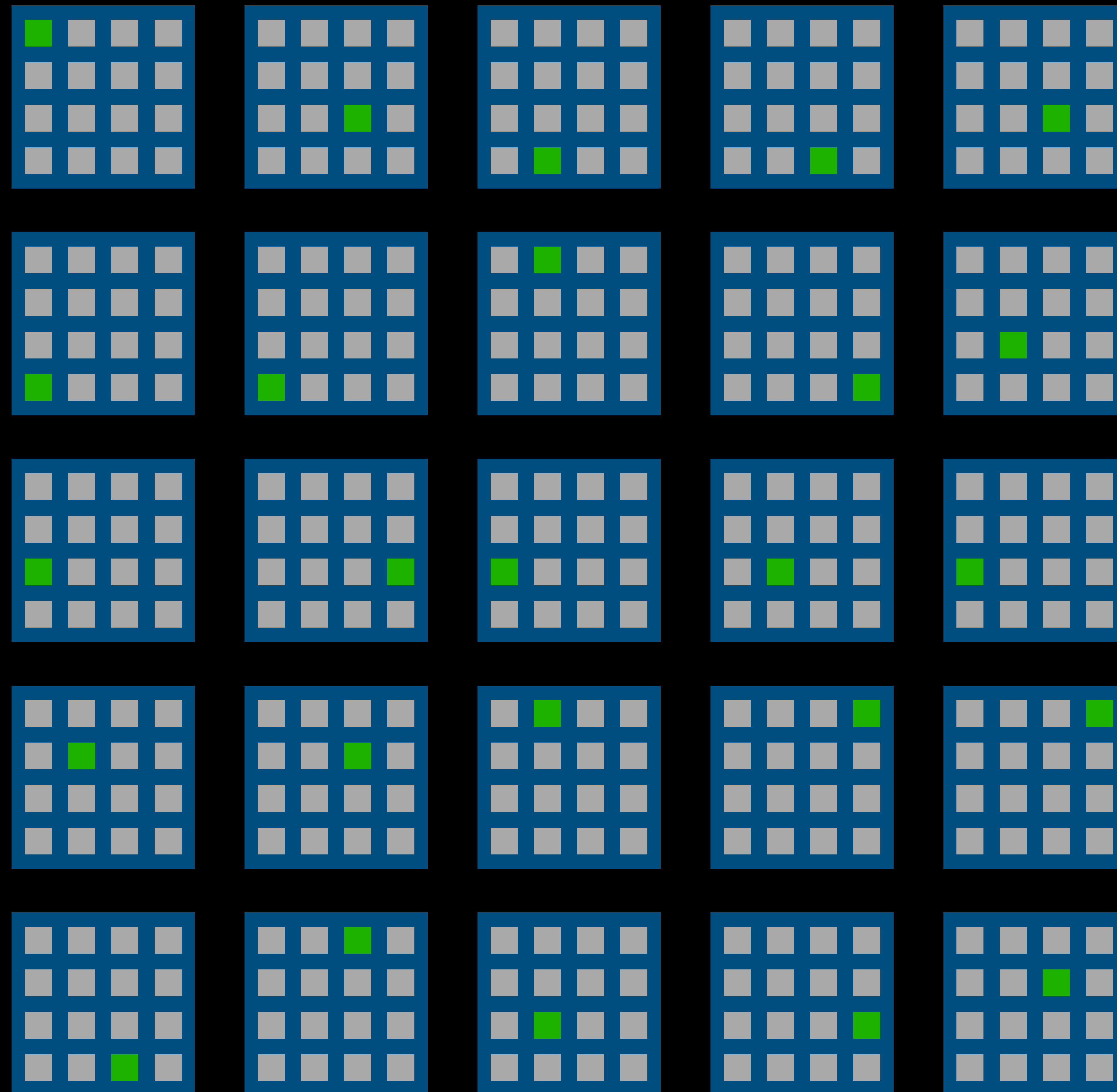
Thread-local shuffle vector

Shuffle Vector: Fast randomized allocation

```
free( )
```



Thread-local shuffle vector



All
Pages
Meshable

Finding pages to Mesh

Problem: Find meshing that releases maximum number of pages

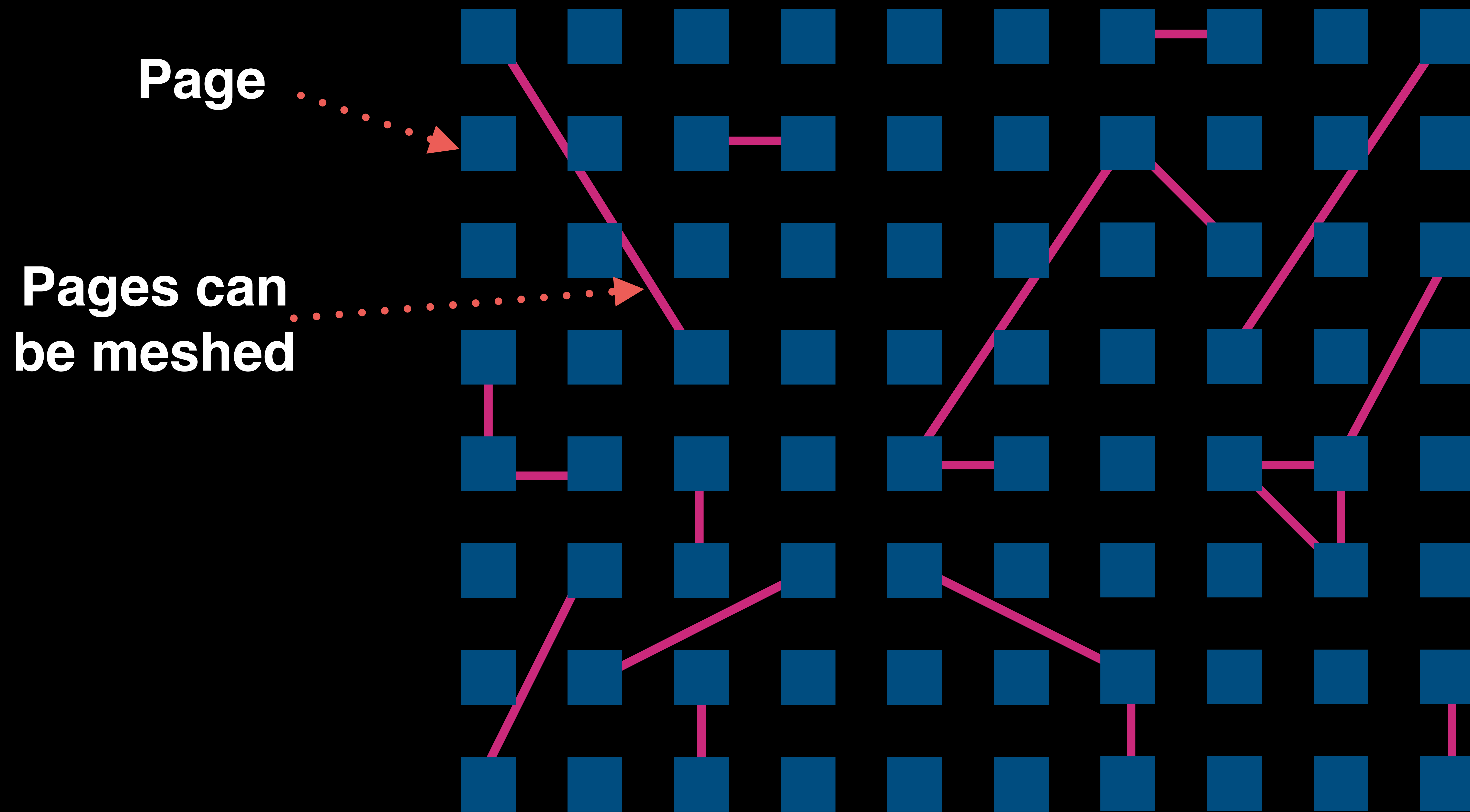
Problem: Find meshing that releases maximum number of pages

Run in `free()` slowpath

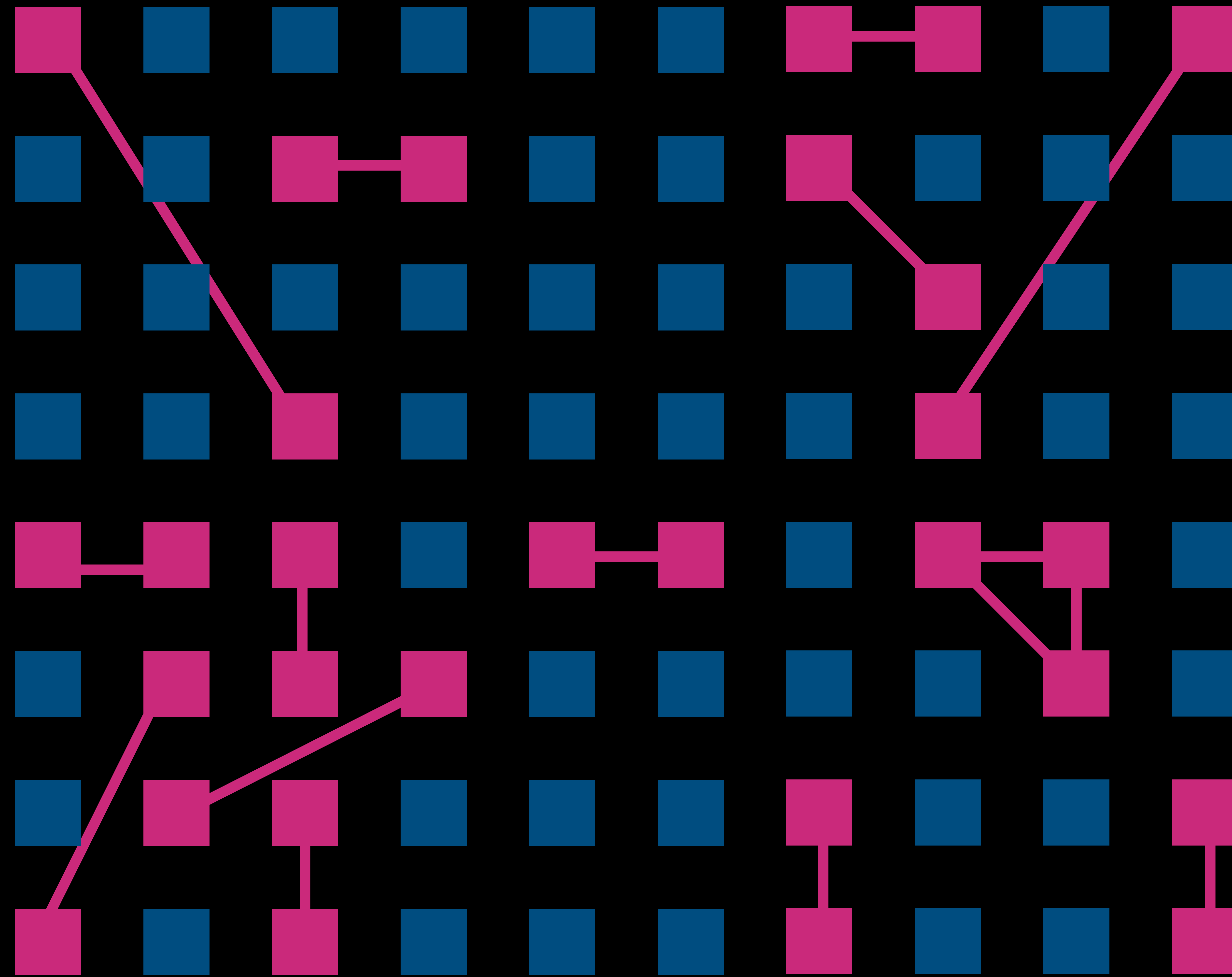
At most once every 100 ms

Treat each size class independently

Problem: Find meshing that releases the maximum number of pages



MinCliqueCover



MinCliqueCover

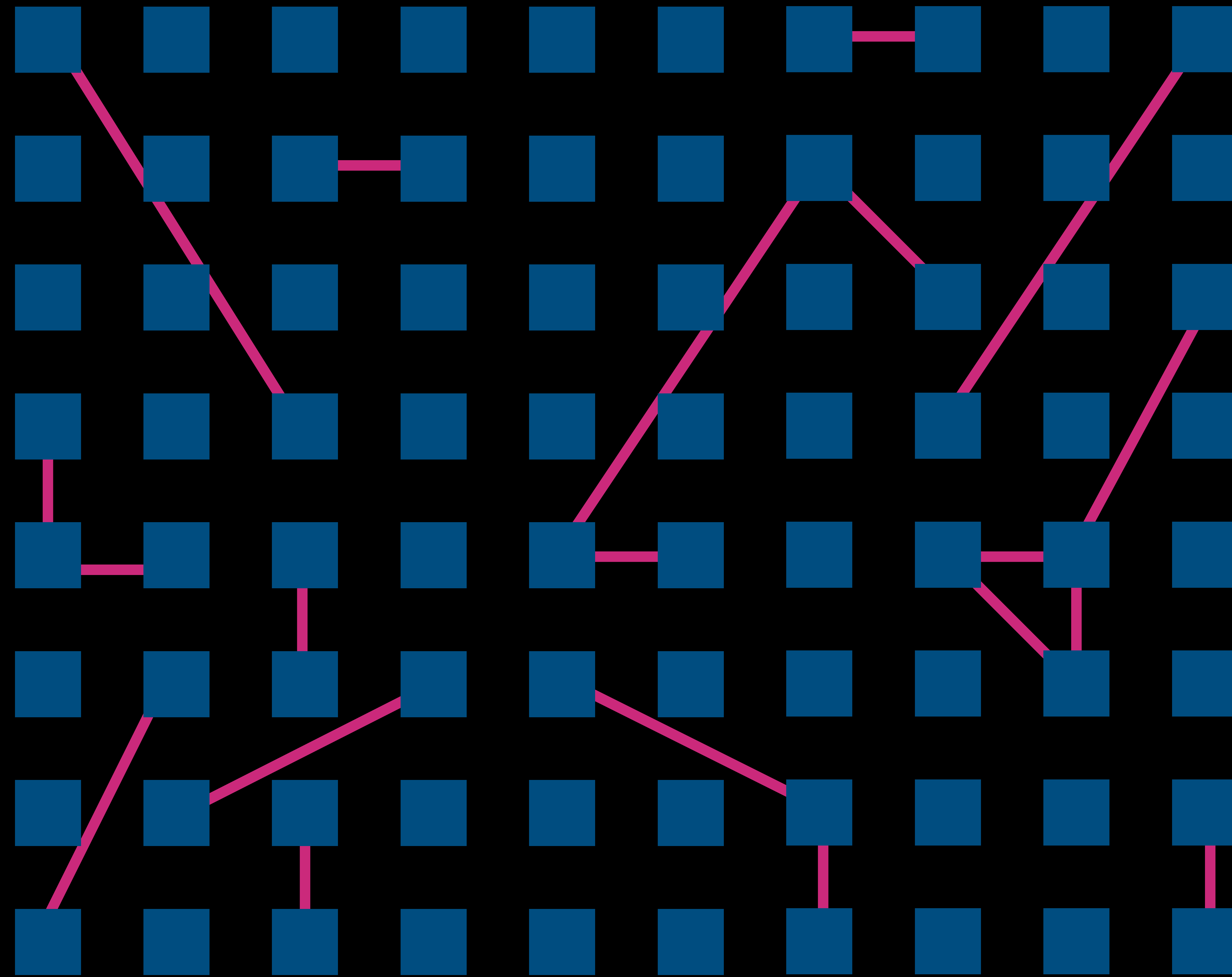
(NP-Complete)

MinCliqueCover

(NP-Complete)

BUT! Randomness ensures we can get away with solving simpler graph problem (Matching)

Wrinkle: building this graph would require RAM + time

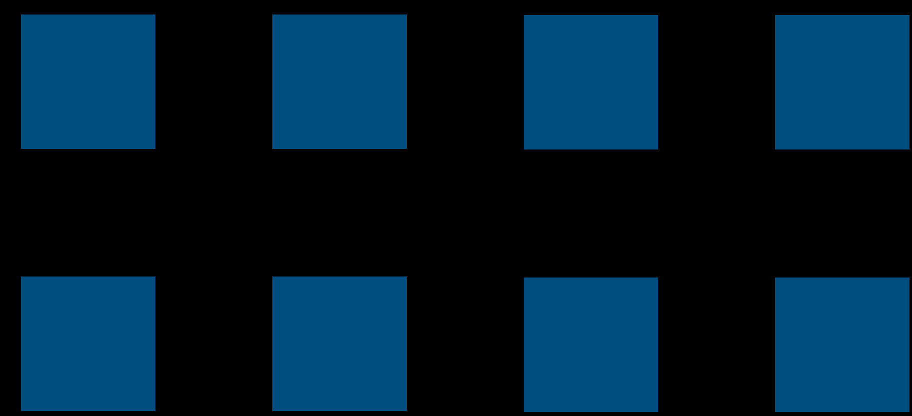


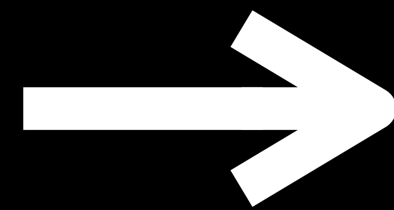
SplitMesher: approximates **Matching** without materializing meshing graph

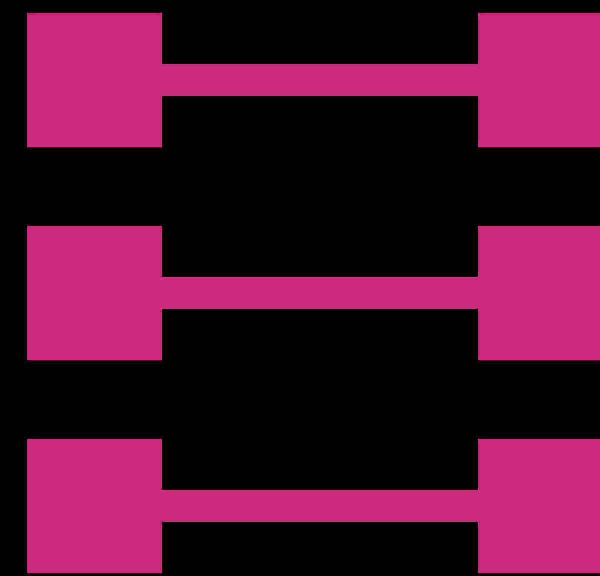
SplitMesher: approximates **Matching** without materializing meshing graph

Set of partially full pages

Pairs of meshable pages

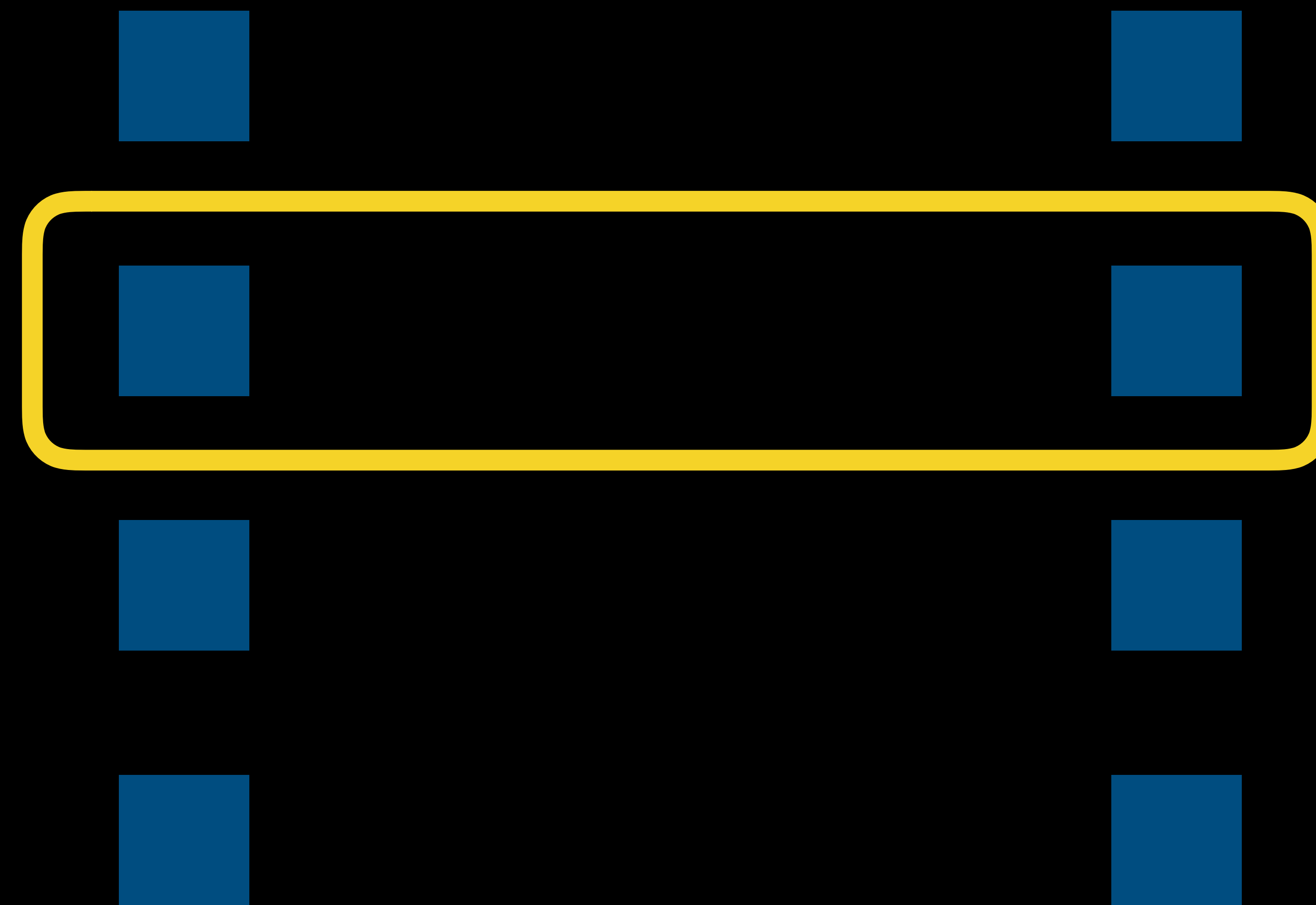




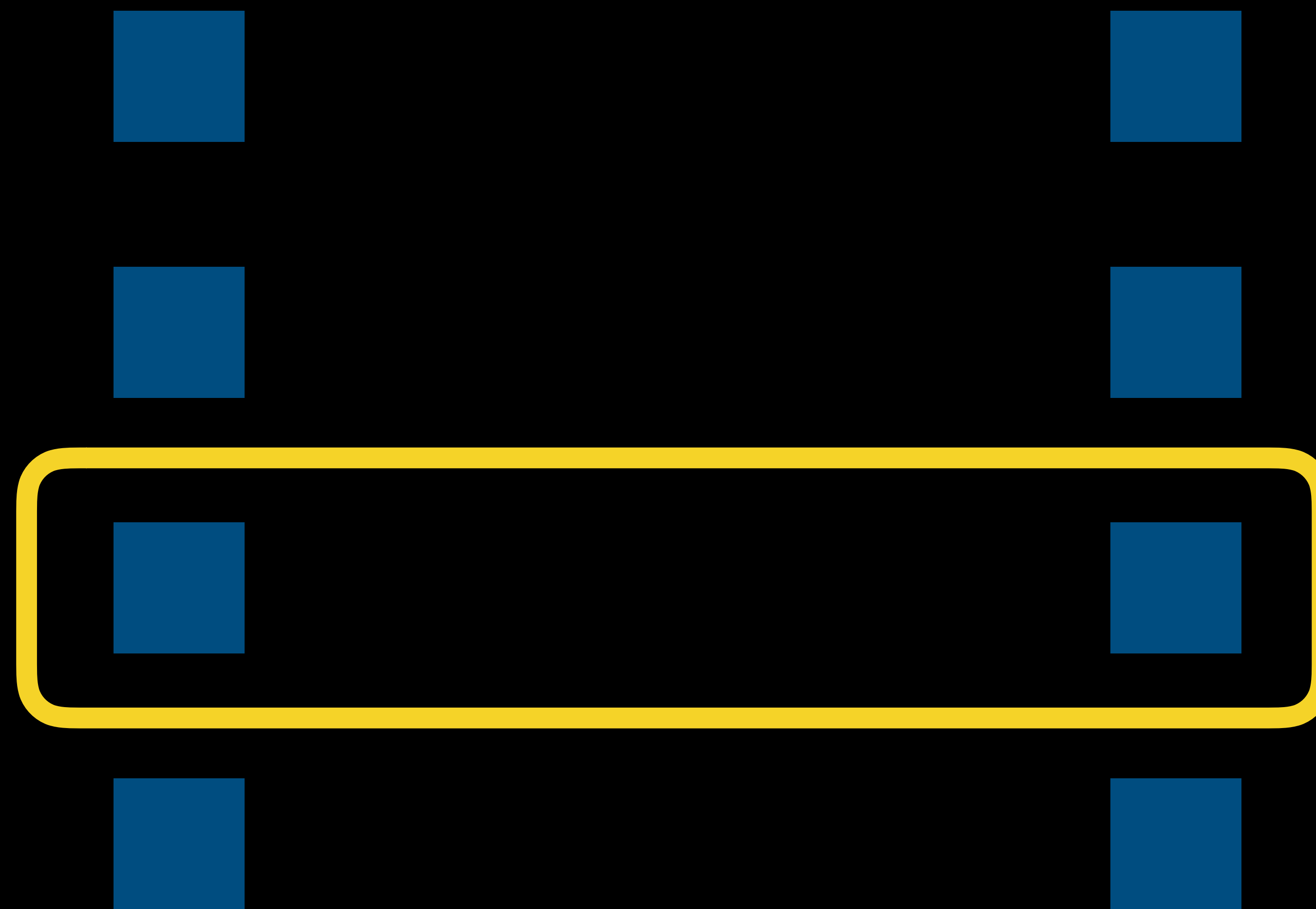


Iterate, comparing a[i] to b[i]

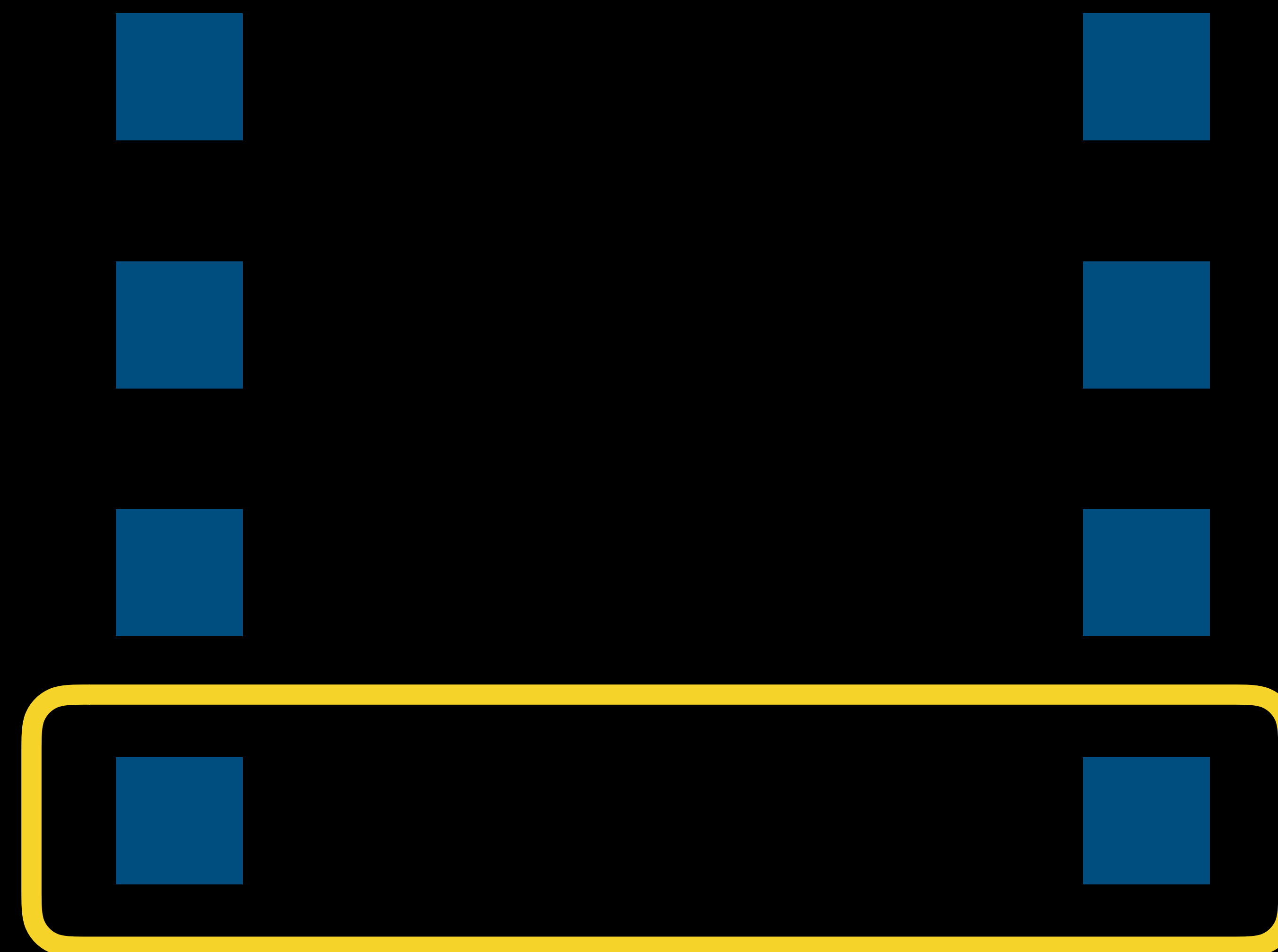
Iterate, comparing a[i] to b[i]



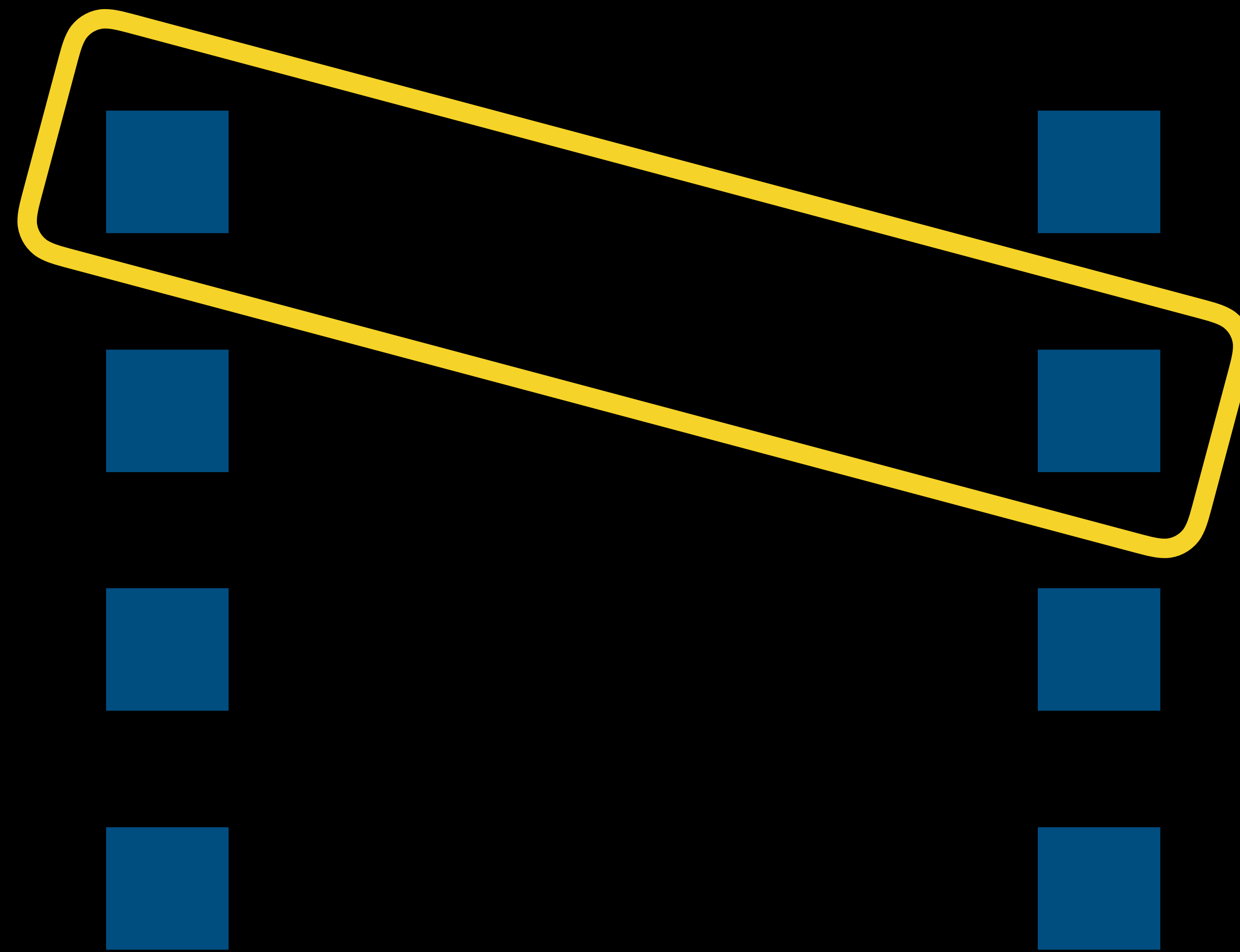
Iterate, comparing a[i] to b[i]



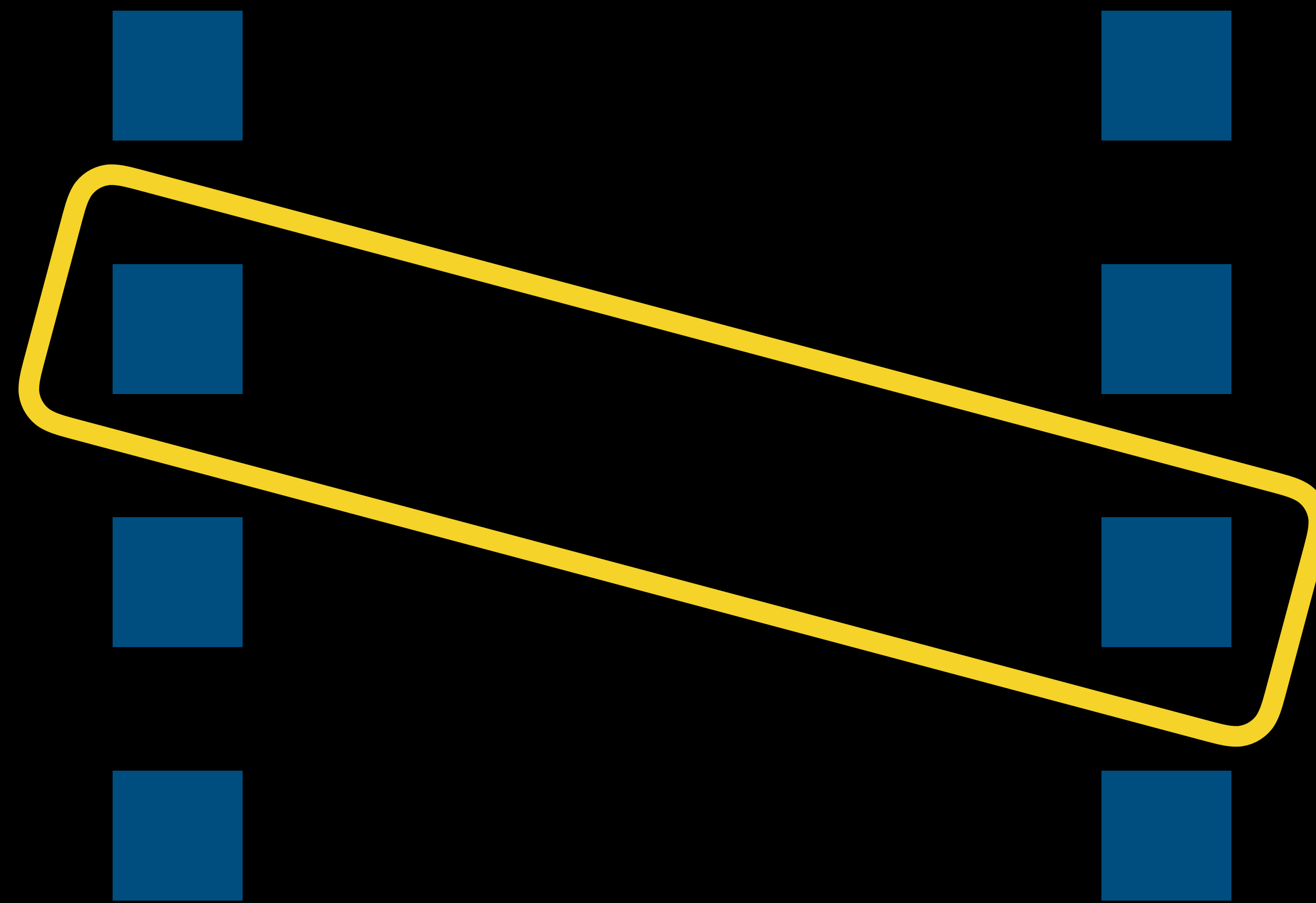
Iterate, comparing a[i] to b[i]

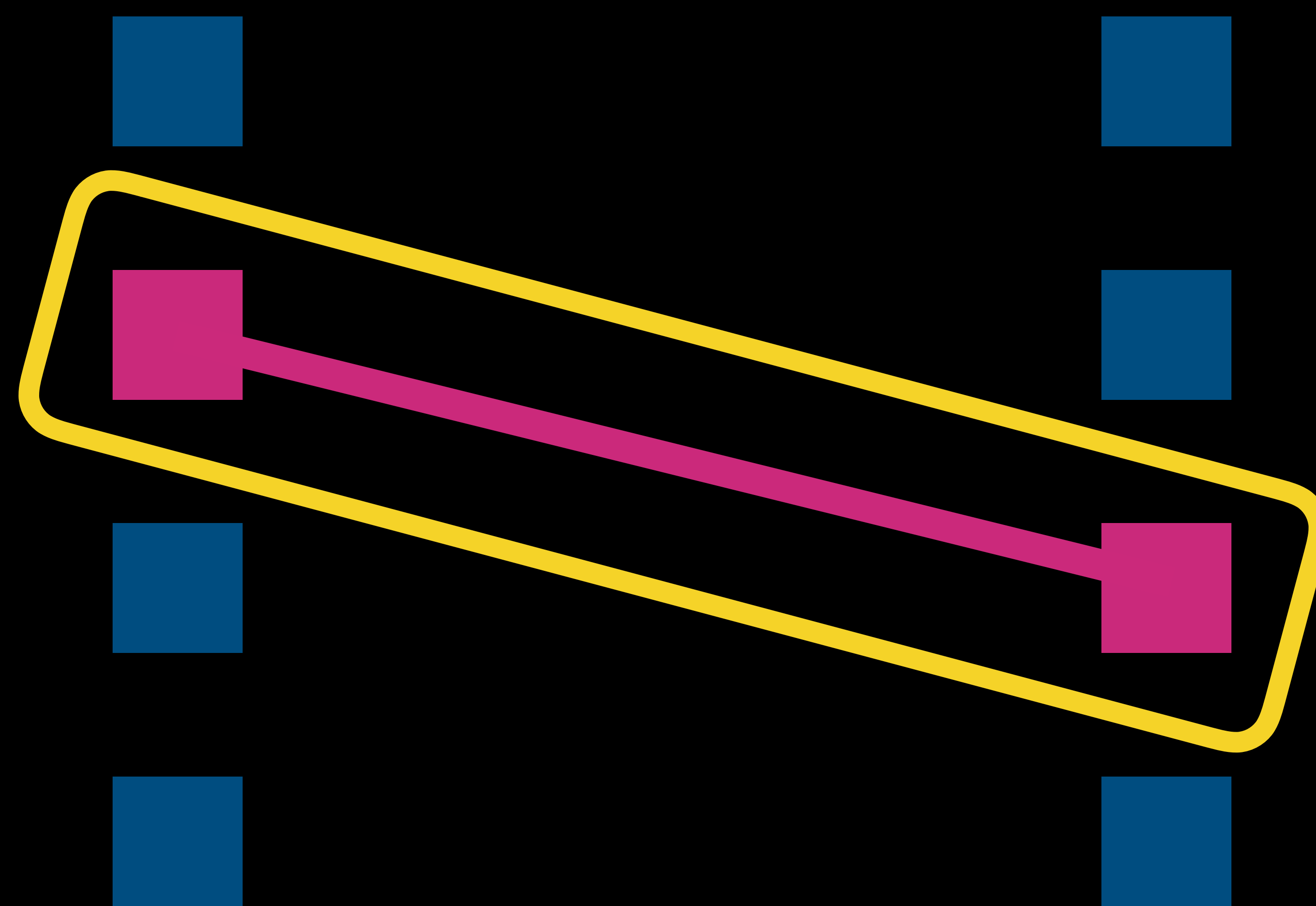


loop, comparing a[i] to b[(i+1)%len]

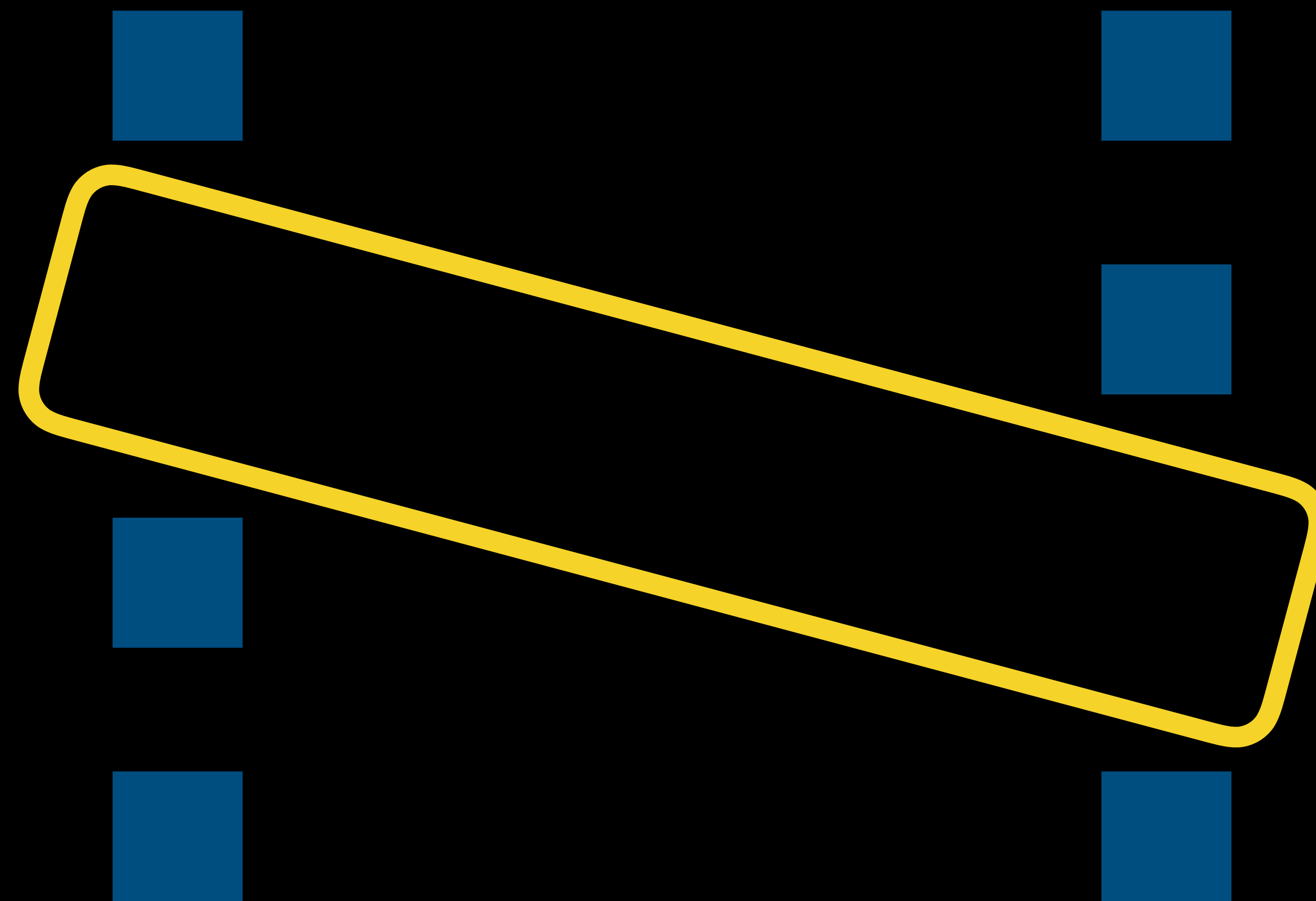


loop **fixed** number of times,
comparing a[i] to b[(i+1)%len]

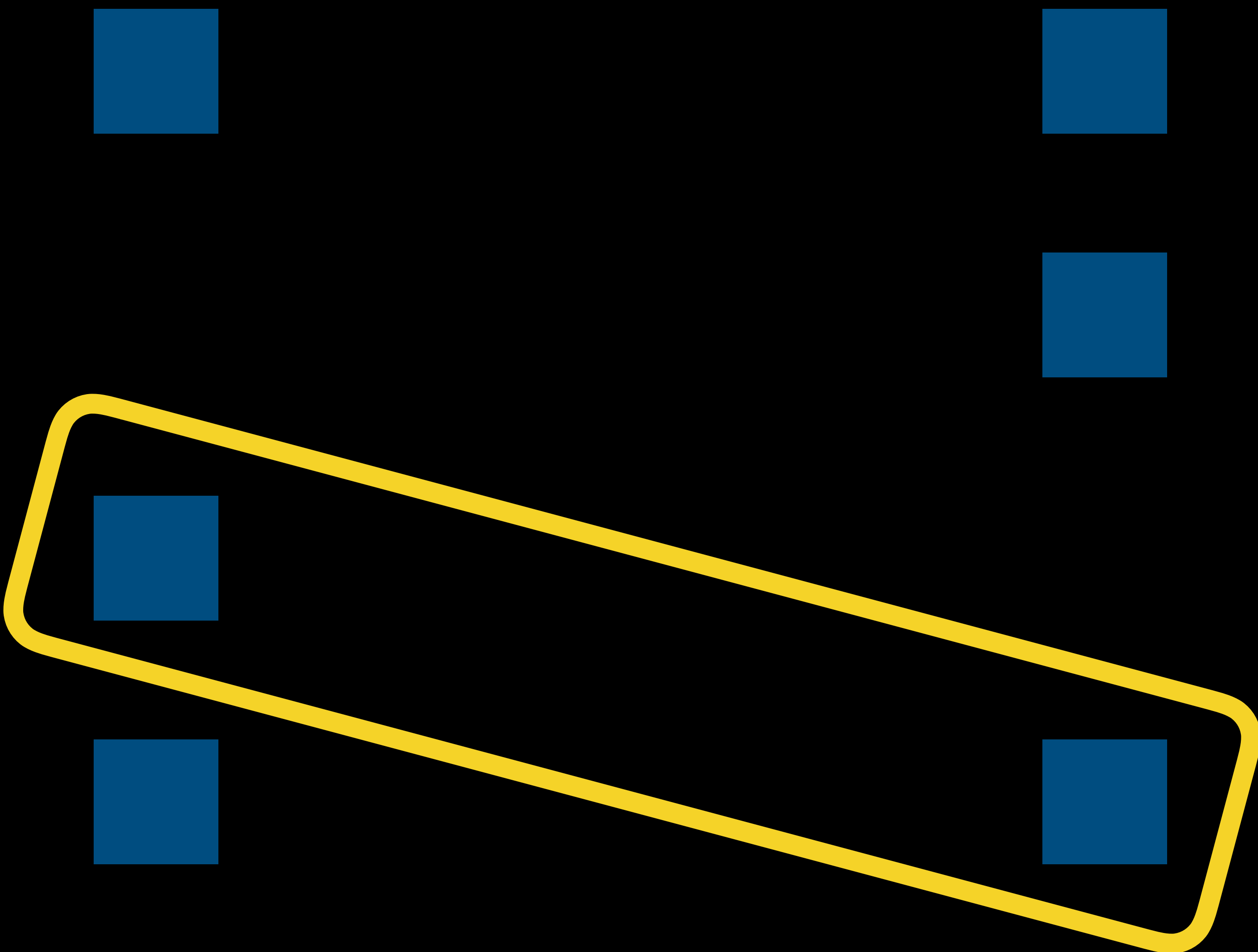




Remove found match



Continue



SplitMesher: approximates **Matching** without materializing meshing graph

SplitMesher: approximates **Matching** without materializing meshing graph

$O(n/q)$ time

(q is the global probability of spans meshing)

SplitMesher: approximates **Matching** without materializing meshing graph

$O(n/q)$ time

(q is the global probability of spans meshing)

1/2* approximation w.h.p.

MESH implementation details

MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

fast framework for building custom allocators!

MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++

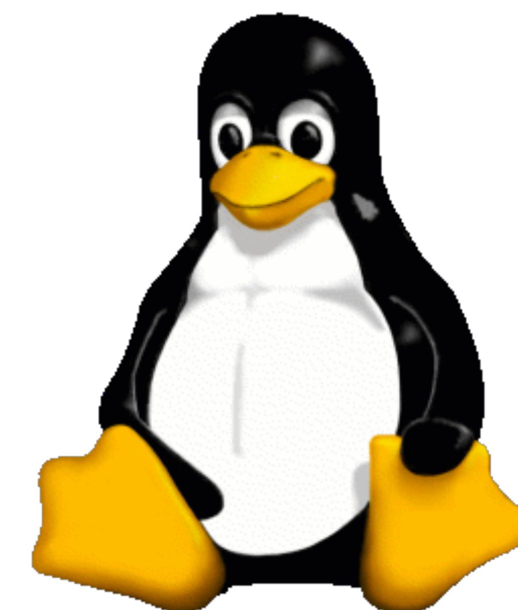
MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++



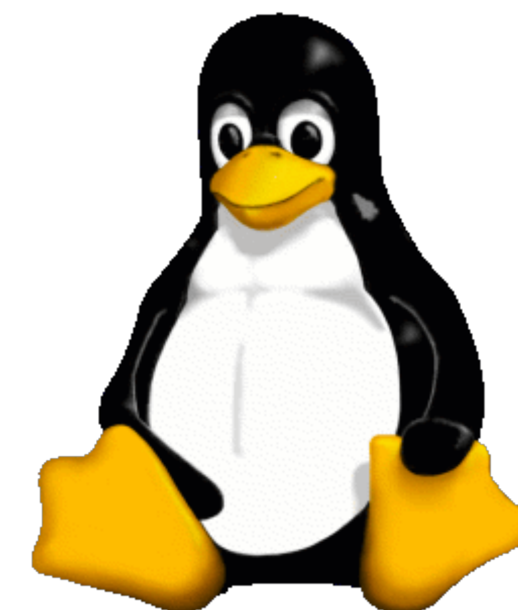
MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++



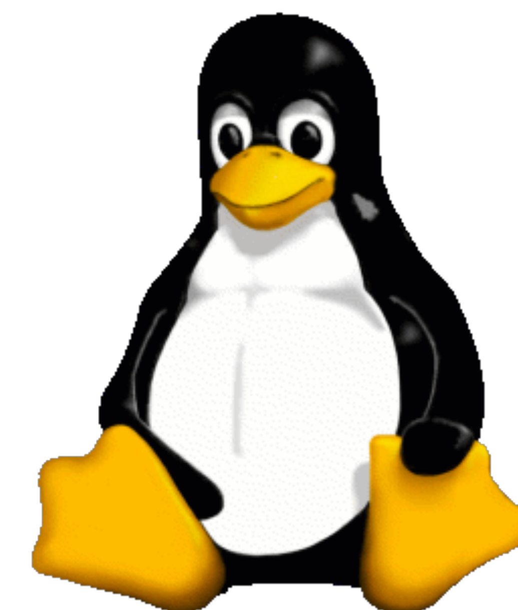
MESH implementation details

built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++



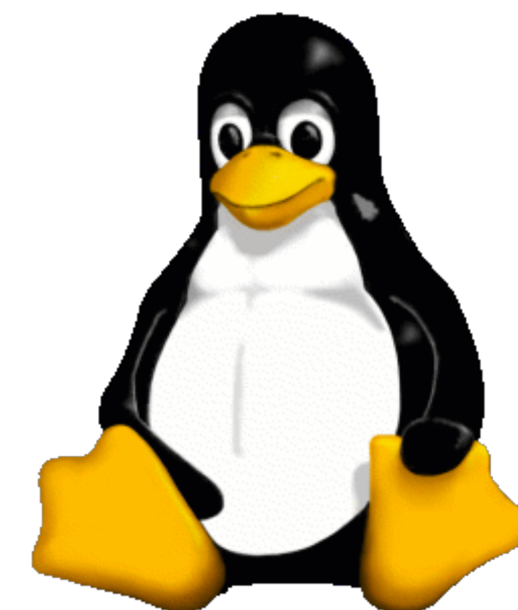
MESH implementation details

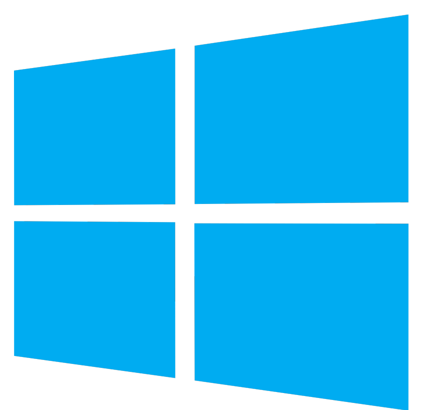
built using **Heap Layers**

github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++





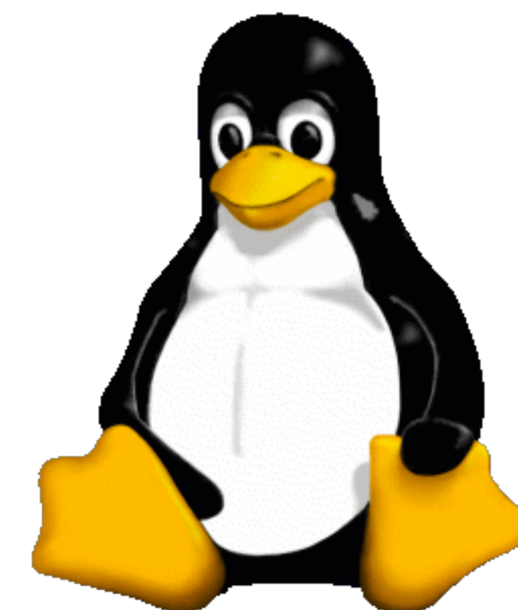
MESH implementation details

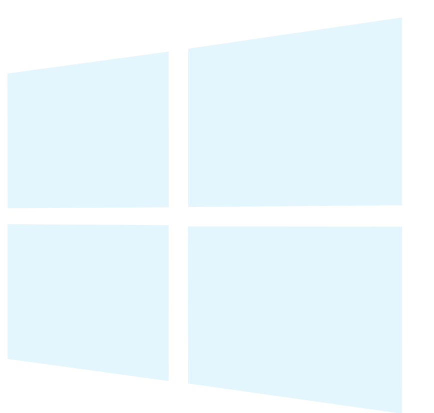
built using **Heap Layers**

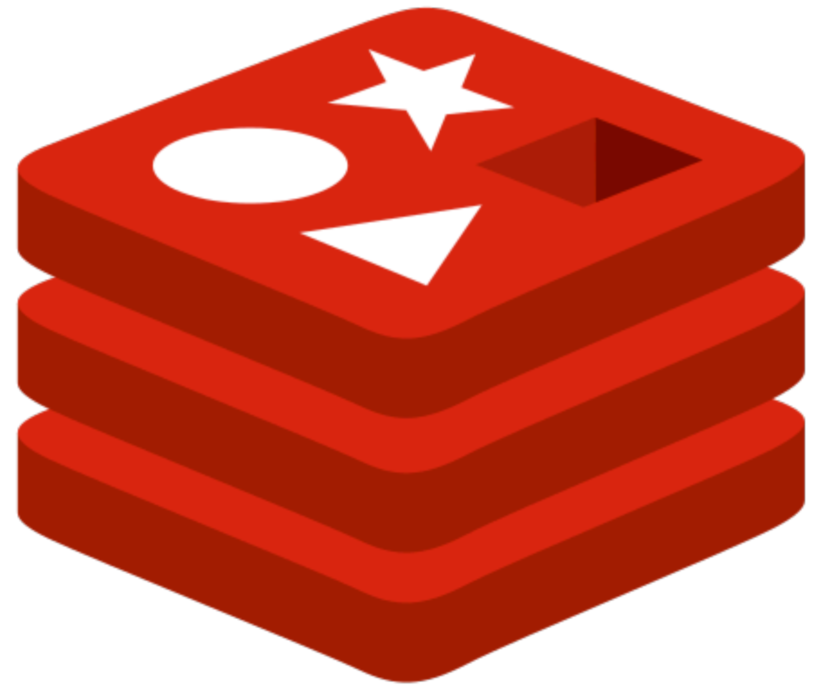
github.com/emeryberger/Heap-Layers

(used to build Hoard, DieHard, DieHarder…)

< 5K SLOC, C++







redis

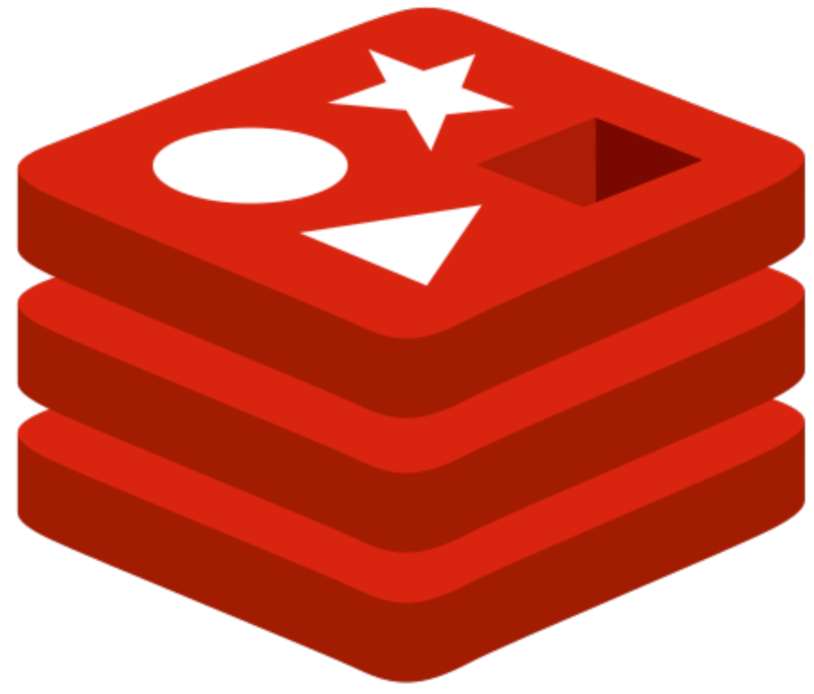
jemalloc + activedefrag
Mesh
no compaction

RSS (MiB)

300
200
100
0

0 1 2 3 4 5 6

Time Since Program Start (seconds)



redis + MESH

17% heap size reduction

< 1% performance overhead

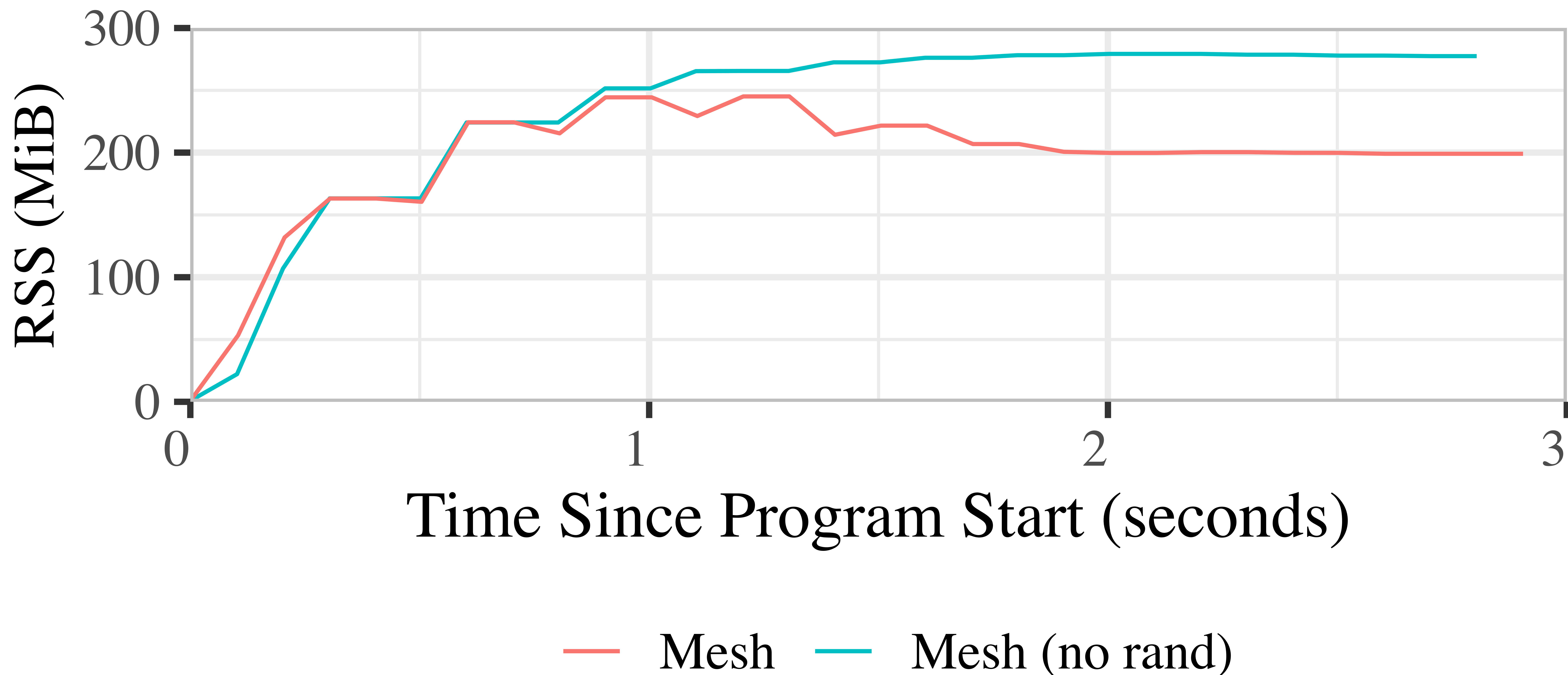
RSS (MiB)

800
600
400
200
0

0
50
100

Time Since Program Start (seconds)

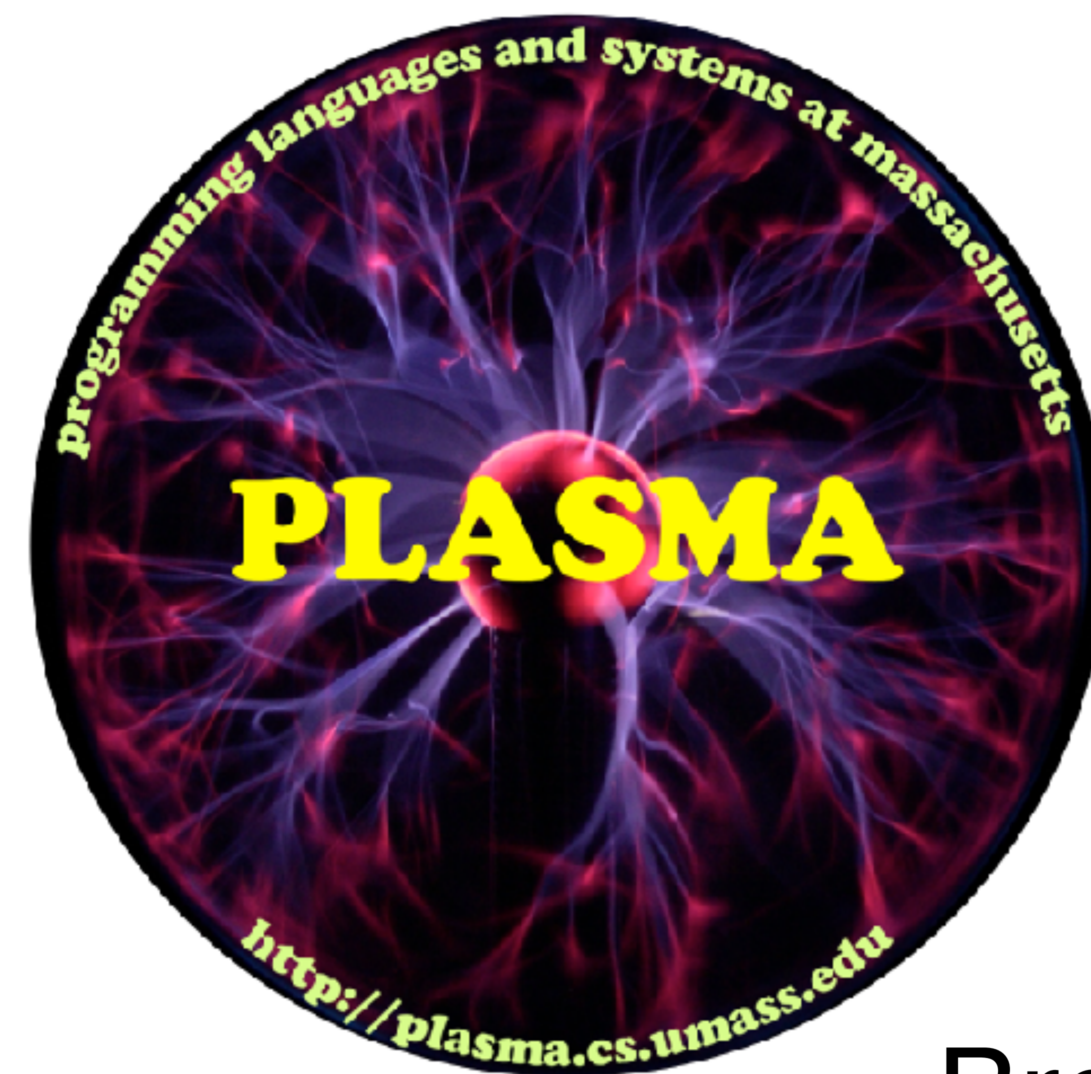
—default jemalloc —Mesh



Ruby Compaction for Free
RSS (MiB)
300
200
100
0
0
1
2
3
Time Since Program Start (seconds)
Mesh
Mesh (no rand)

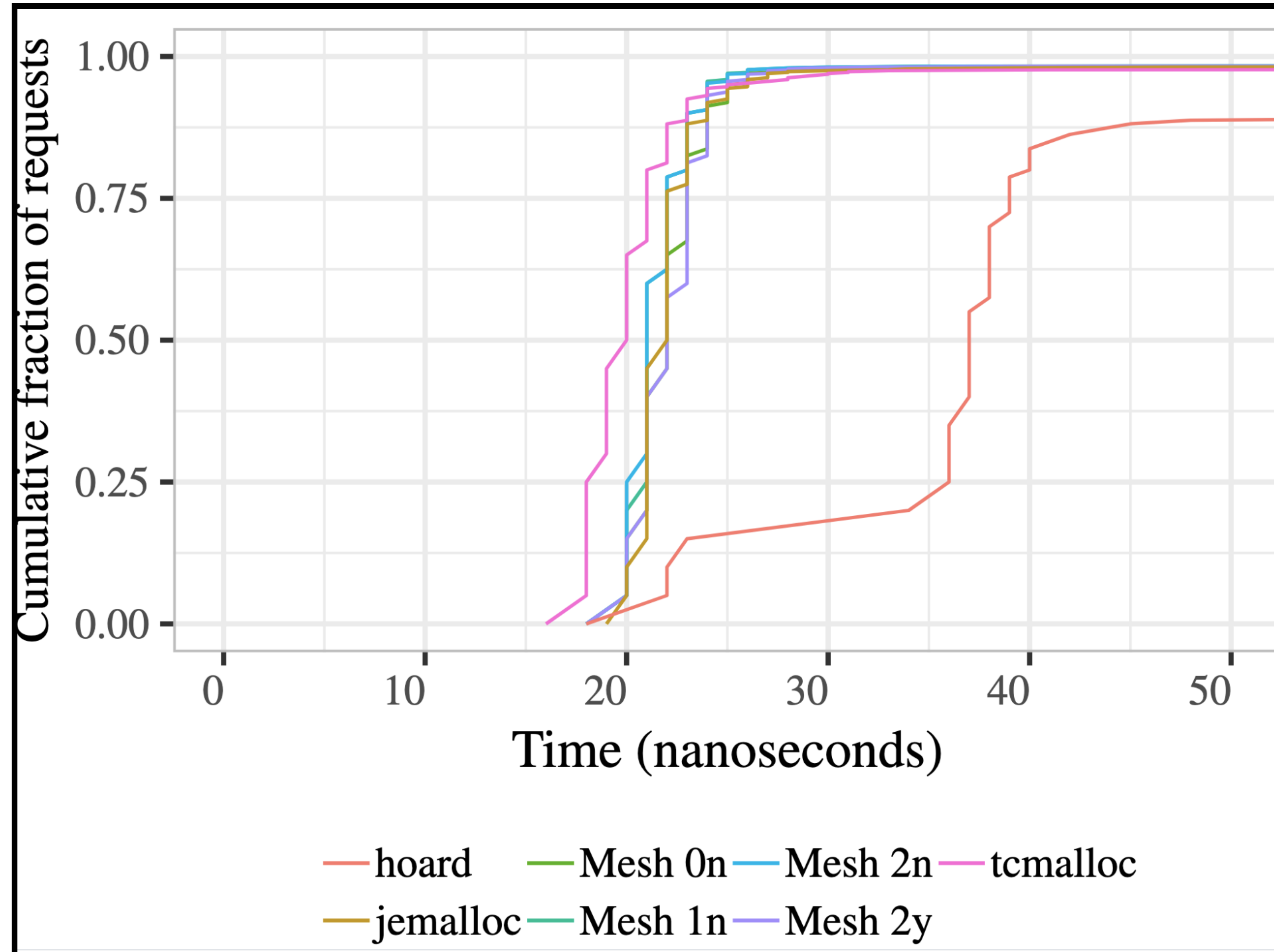
http:// LIBMESH .org

Compaction without relocation!



Browsix, Coz, DieHard, Hoard…

Latency: malloc



Latency: free