Research Topics in Database Management

Bigger, Faster, and Stronger Systems

Niv Dayan

Who am I?

> 12 years of research experience in data structures & algorithms for databases

https://www.nivdayan.net/

Who am I?

> 12 years of research experience in data structures & algorithms for databases

In academia

IT University
of Copenhagen

Who am I?

> 12 years of research experience in data structures & algorithms for databases

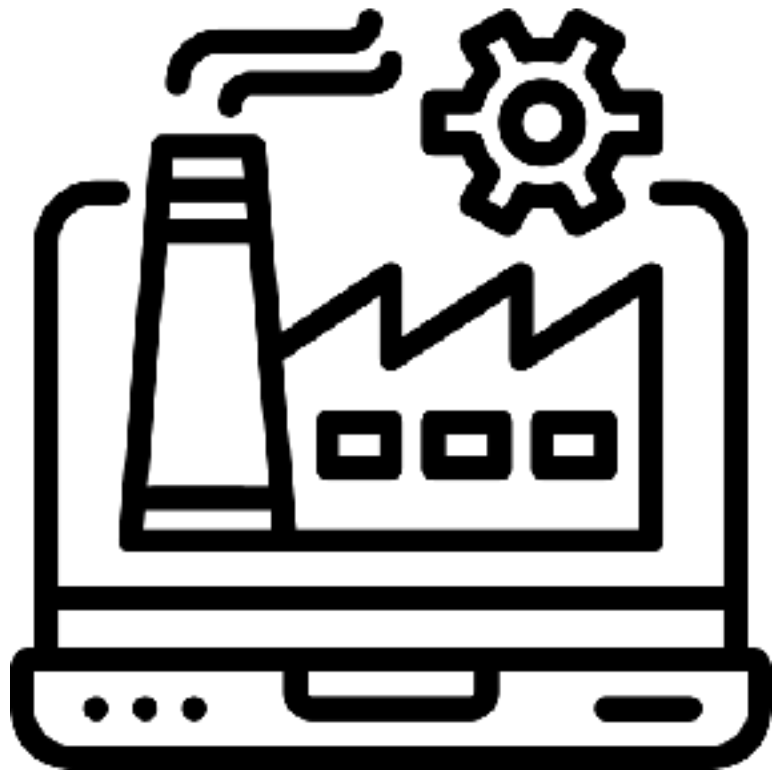
In academia

And in industry

This course combines both

Theory

Practice



For data structures & algorithms for databases

Who are you?

Who are you?

Undergrad

Grad

Who are you?   Prerequisites…

Who are you? Prerequisites…

Operating Systems

Concurrency & synchronization
File systems, virtual memory

Who are you? Prerequisites…

Operating Systems

Concurrency & synchronization
File systems, virtual memory

Design and Analysis of Data Structures

Binary trees, sorting, hash tables, priority queues, Big-O analysis

Who are you? Prerequisites…

Operating Systems

Concurrency & synchronization
File systems, virtual memory

Design and Analysis of Data Structures

Binary trees, sorting, hash tables, priority queues, Big-O analysis

Database Internals e.g., (CSC443)

Storage, buffer pools, B-trees, transactions, write-ahead logging, query processing, etc.

Who are you?   Prerequisites…

Operating Systems

Concurrency & synchronization
File systems, virtual memory

Design and Analysis of Data Structures

Binary trees, sorting, hash tables, priority queues, Big-O analysis

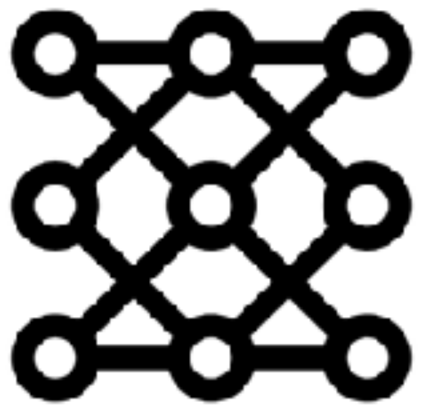
Database Internals e.g., (CSC443)

Storage, buffer pools, B-trees, transactions, write-ahead logging, query processing, etc.

Solid programming skills in C, C++, Java, or at least Python

Background Knowledge

CSC443 is background for some topics



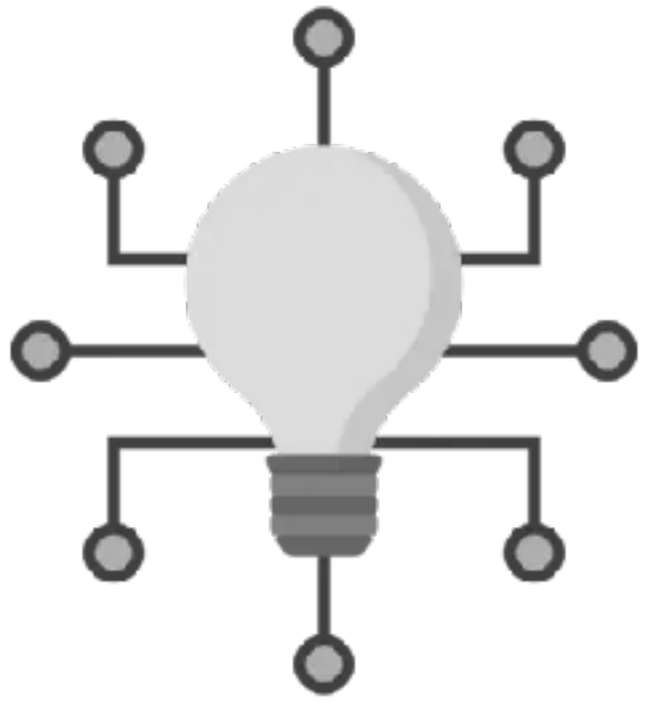
All lectures recorded

Will let you know what to catch up on

https://www.nivdayan.net/database-system-technology-csc443

Data Structures Seminar

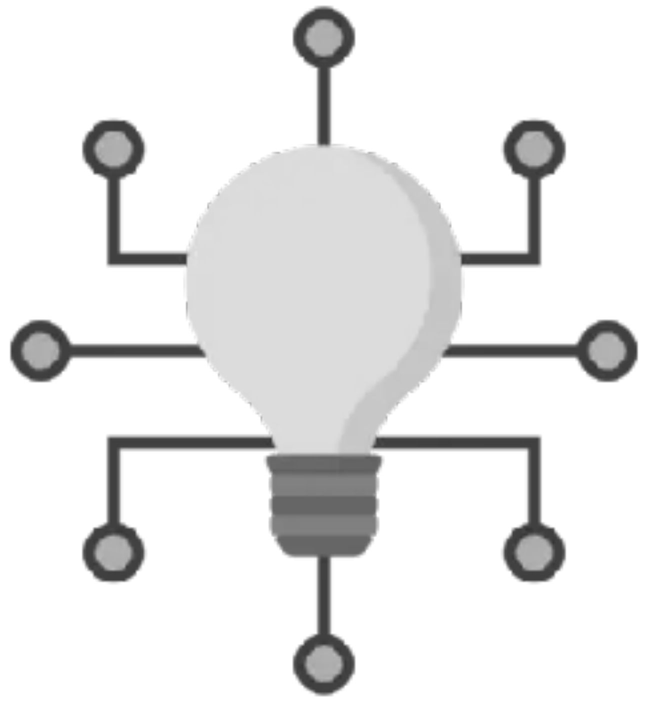
Reading ≈20-30 Papers



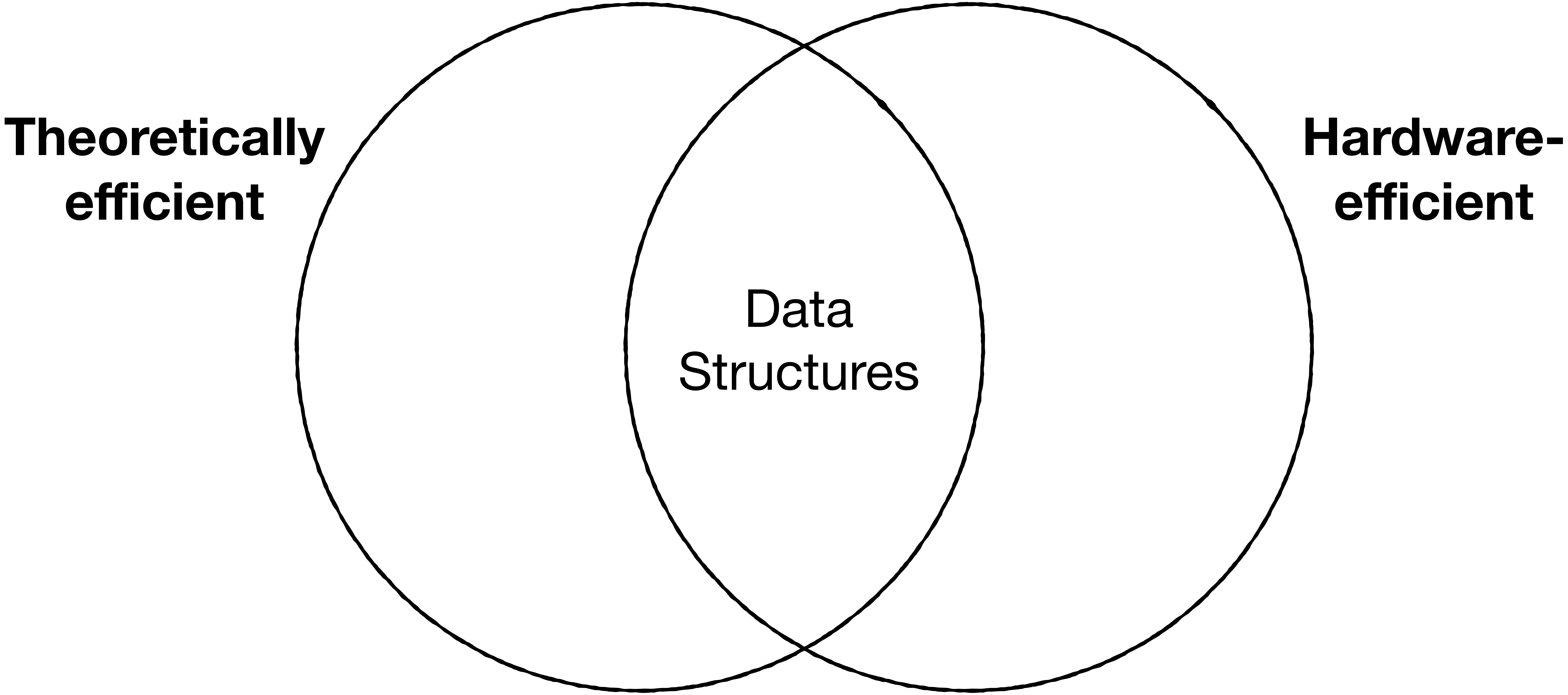
Small Research Project

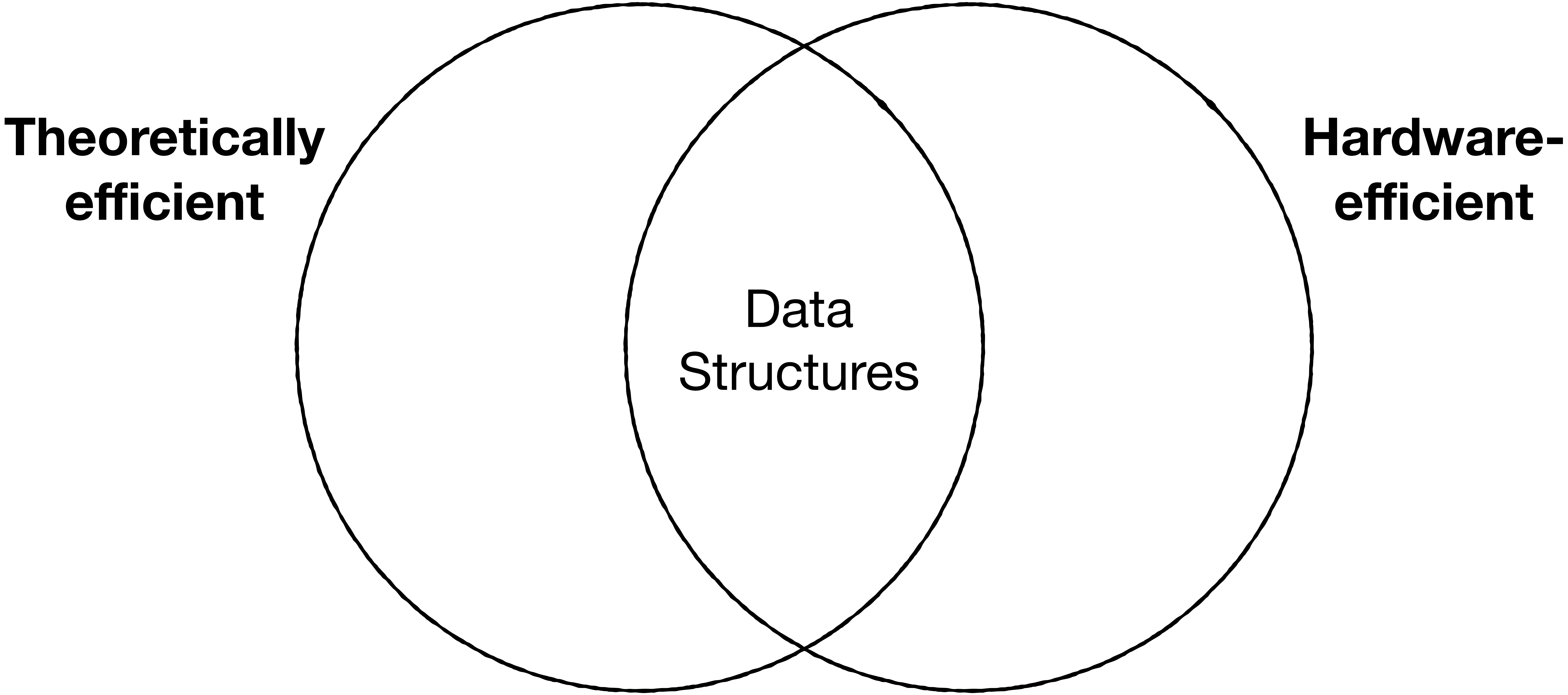
Data Structures Seminar

Reading ≈20-30 Papers



Small Research Project


Theoretically efficient
Hardware-efficient
Data Structures



Important for your maturity as engineers/researchers who can achieve high performance

Why read papers?

Reading papers is a skill

Get research ideas

Employ the state of the art

Website

https://www.nivdayan.net/research-topics-in-database-management-csc2525

12 Class Sessions

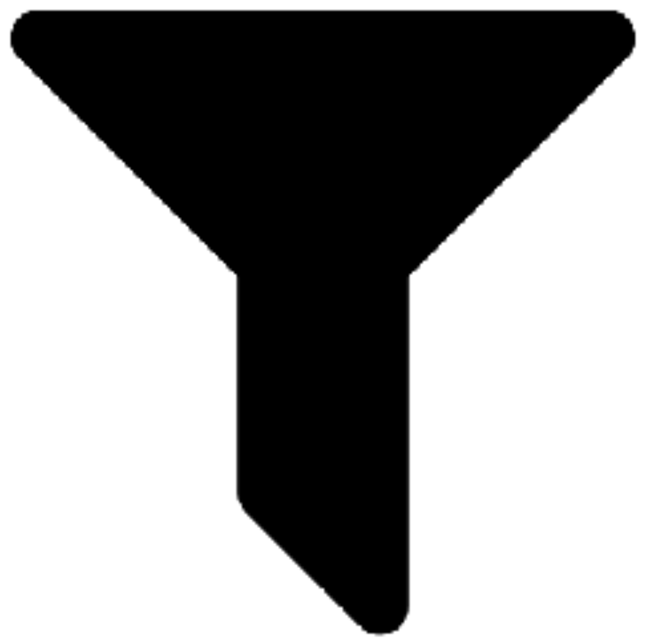
Use the first two lectures wisely

Dynamic Arrays &
Filter Data
Structures

Enjoy the material?
you’re in the right
place

Use the first two lectures wisely

Dynamic Arrays &
Filter Data
Structures



Enjoy the material?
you’re in the right
place

Participation

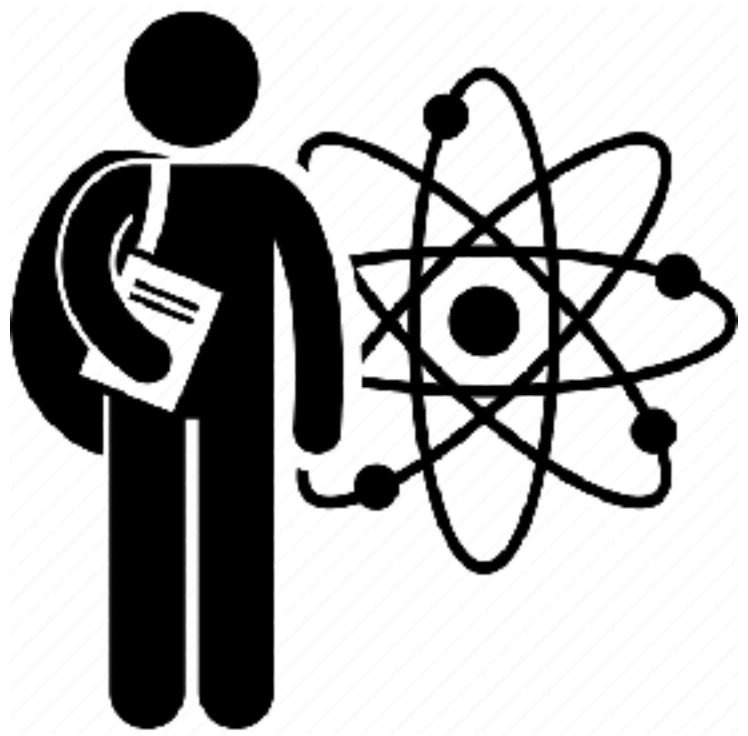
You are required to attend each class

Read papers in advance

Participate in class discussions

Project

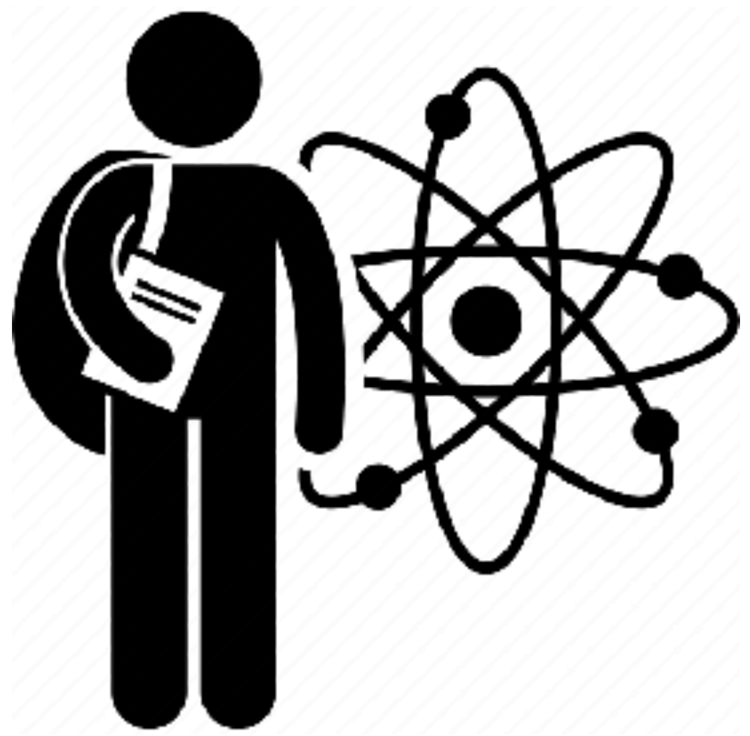
Implement & evaluate



Proposals due by mid-Feb

Project

Implement & evaluate

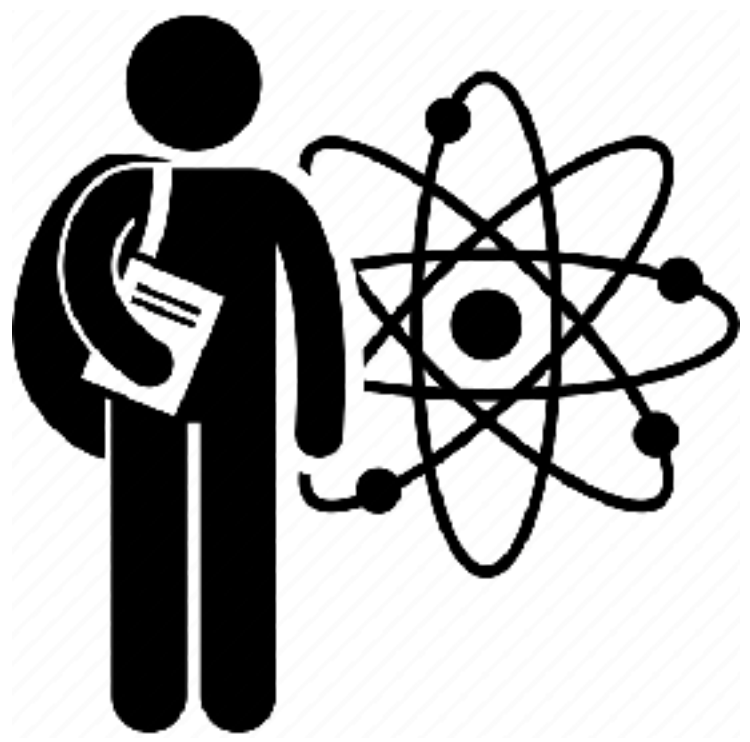


Proposals due by mid-Feb

You may start earlier

Project

Implement & evaluate



Proposals due by mid-Feb

You may start earlier

More on this later

Written Exam

Likely 2 hours

Likely April 7-8 or 30
(Before/after exam period)

Grade Components

(1) Project Report & Code

(2) Oral exam

Grade Components

(1) Project Report & Code

(2) Oral exam

Precise breakdown to be announced later

Office Hours

Right after
class

Post questions for everyone’s benefit!

We’ll record classes, but you must still attend.

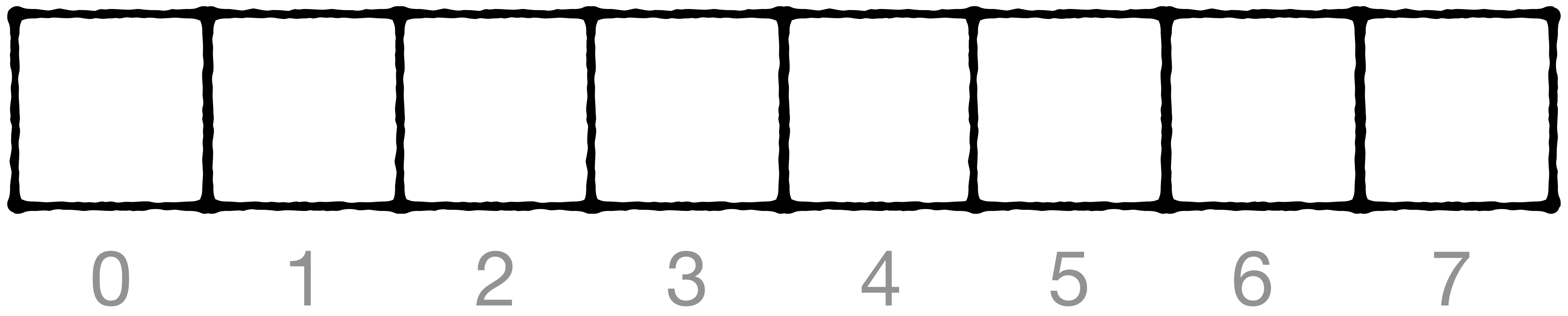
And now to our first lecture

Dynamic Arrays

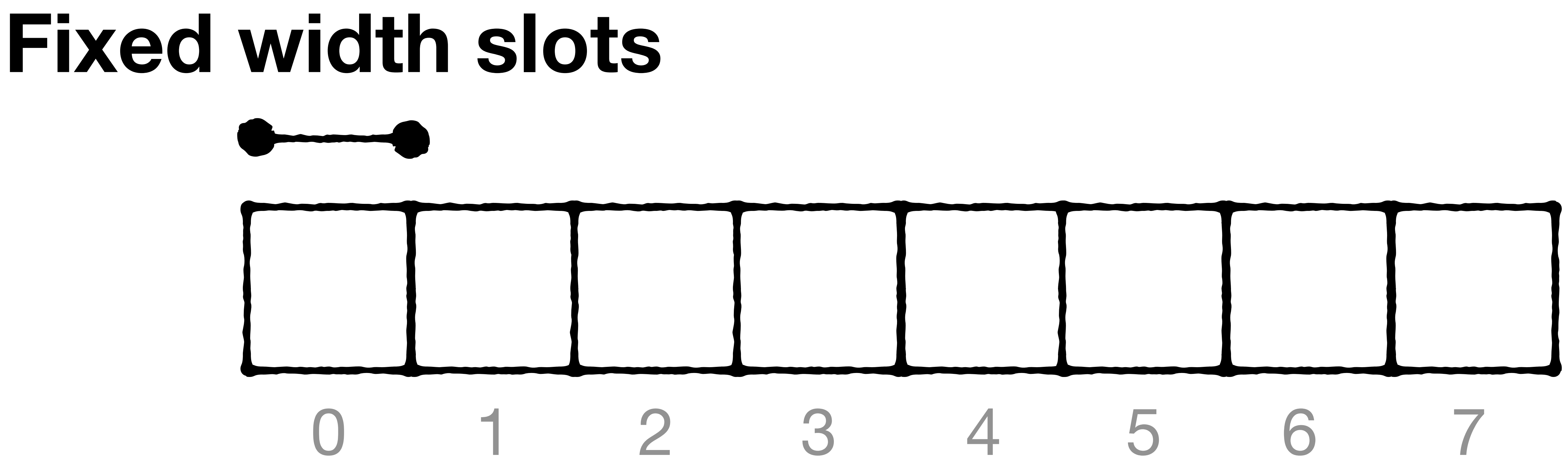
CSC2525 Research Topics in Database Management

Niv Dayan

Arrays

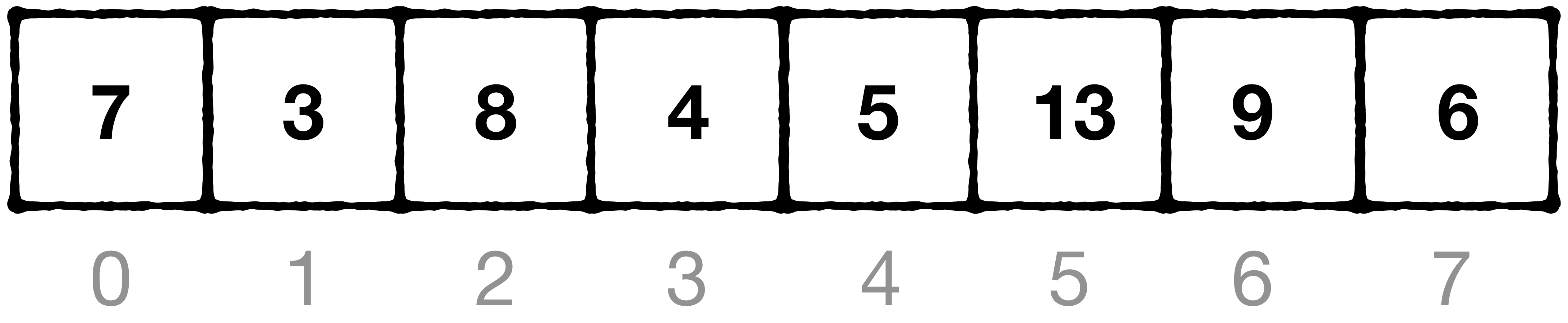


Arrays



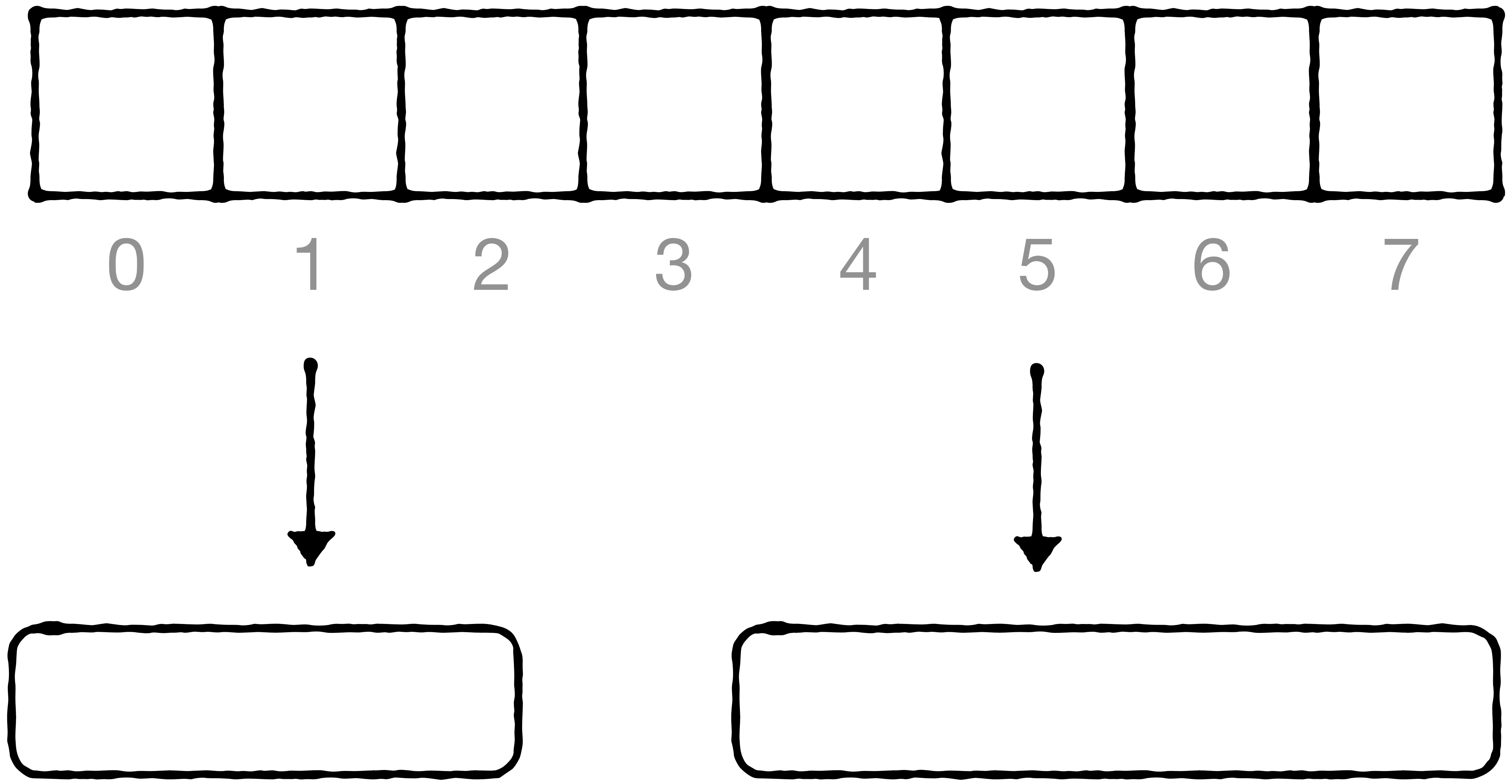
Fixed width slots

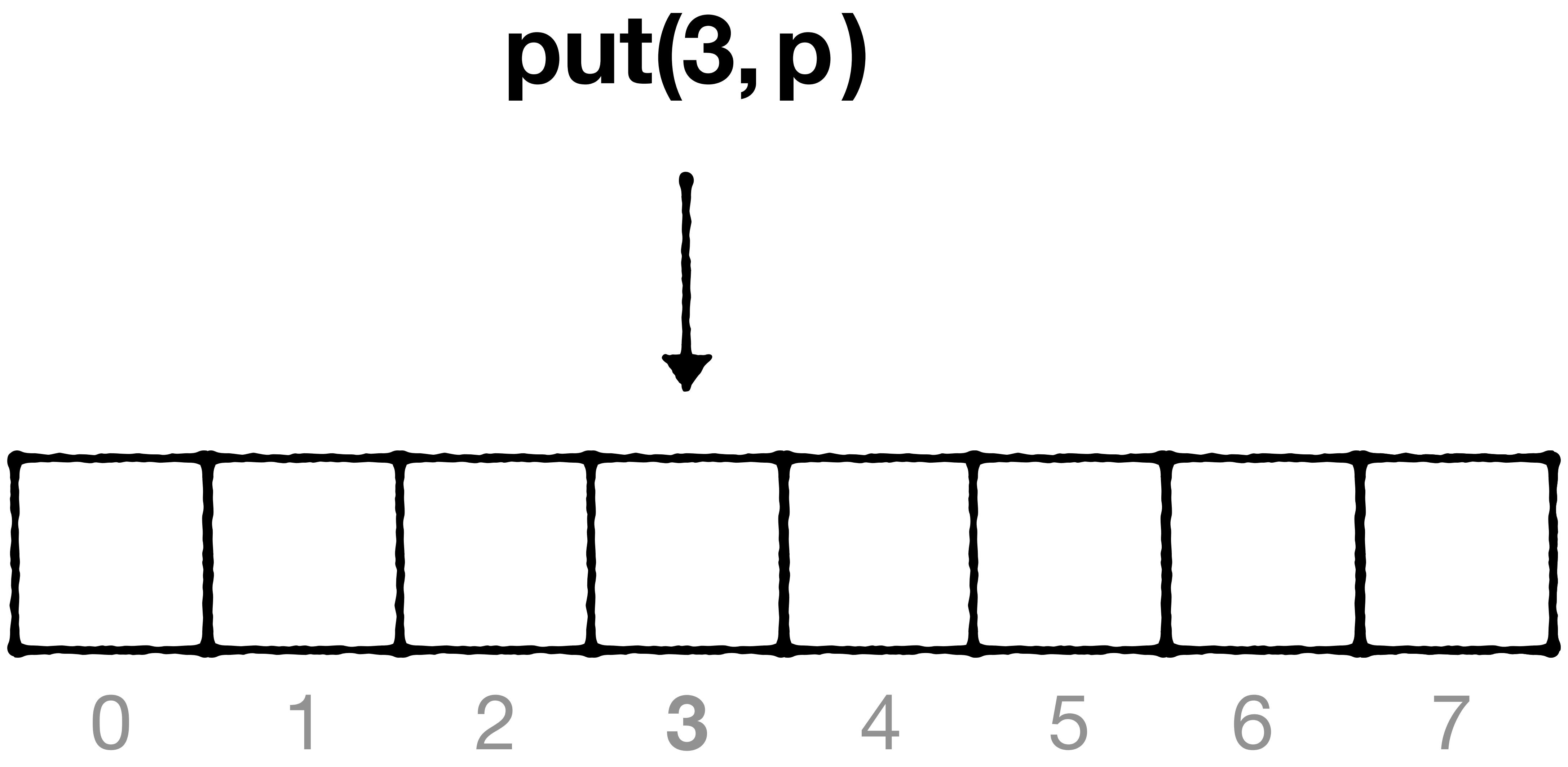
e.g., integers or floating points



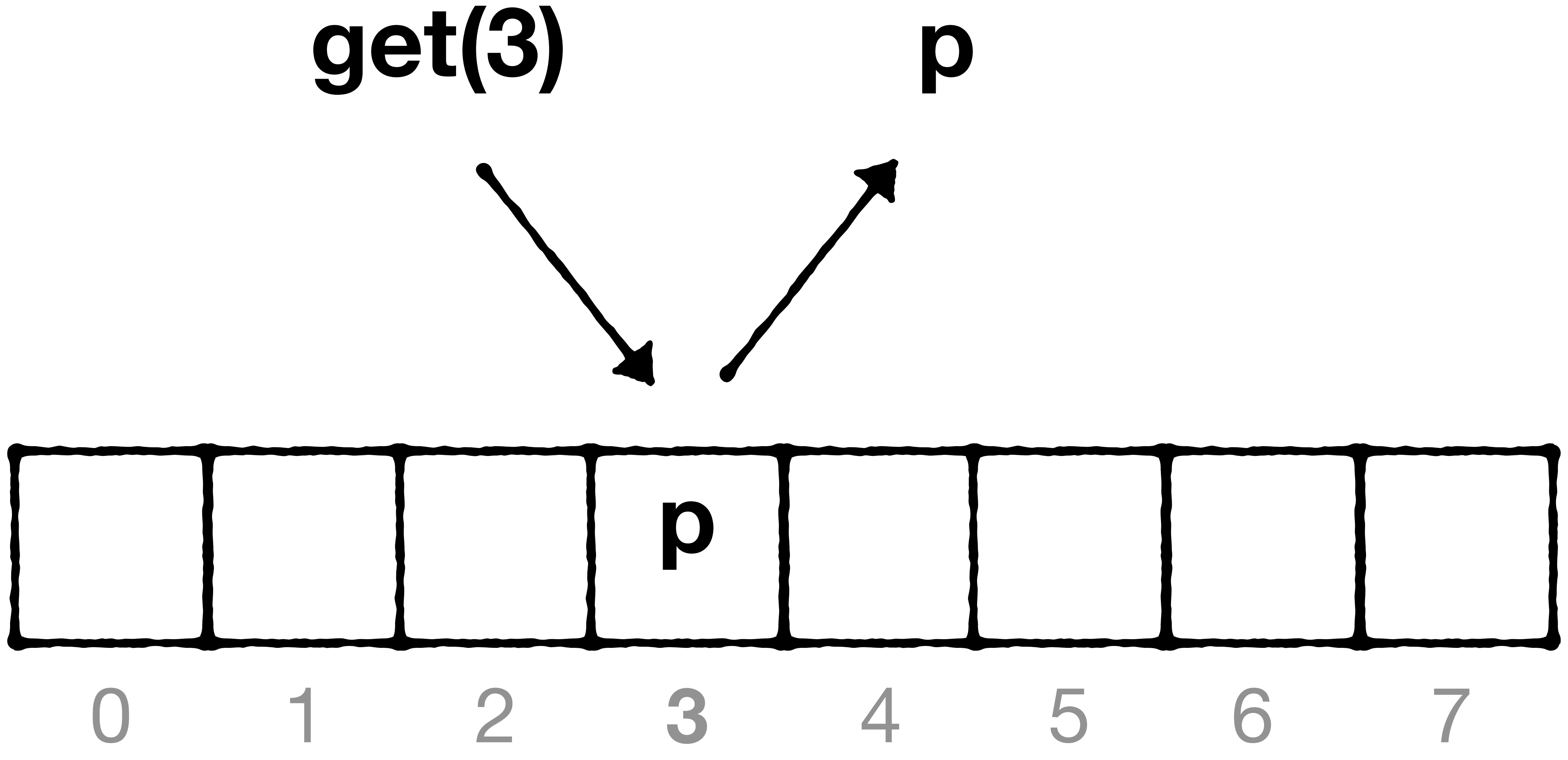
Fixed width slots

Or pointers to complex types

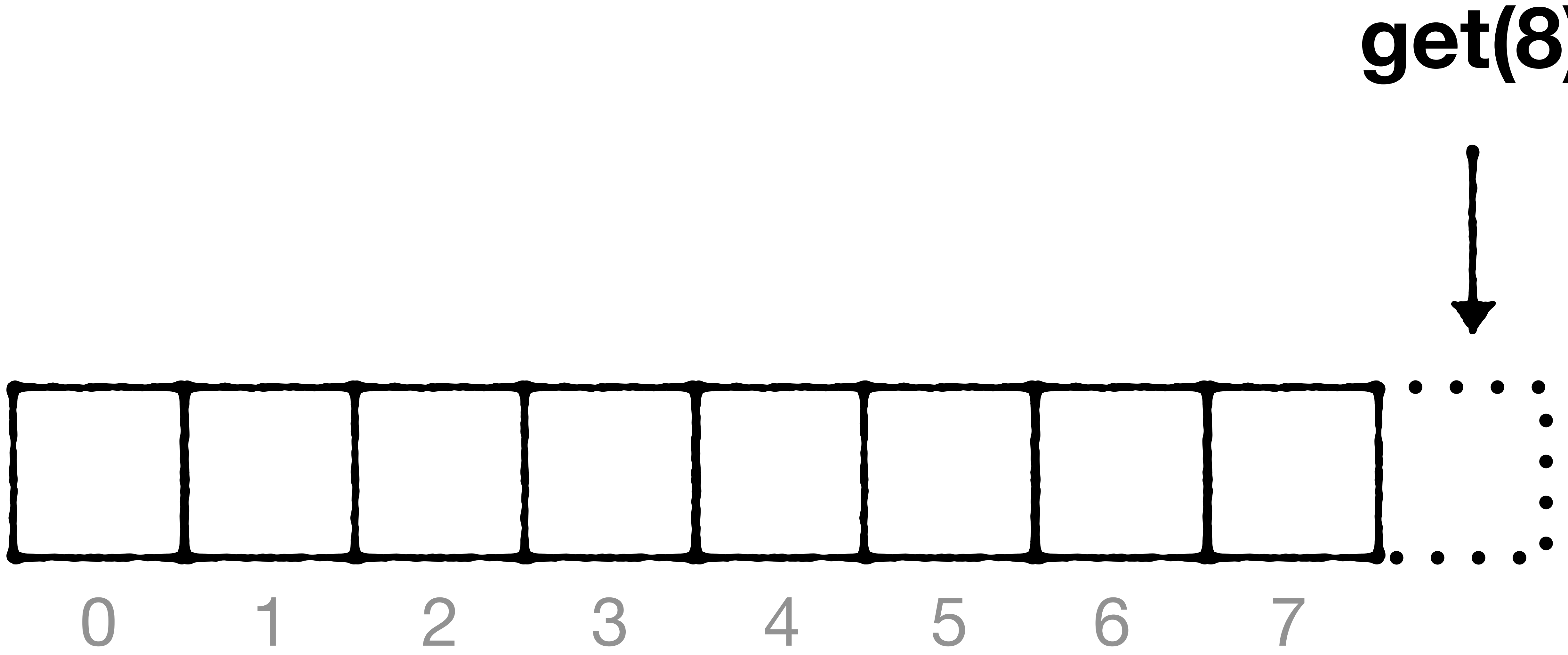




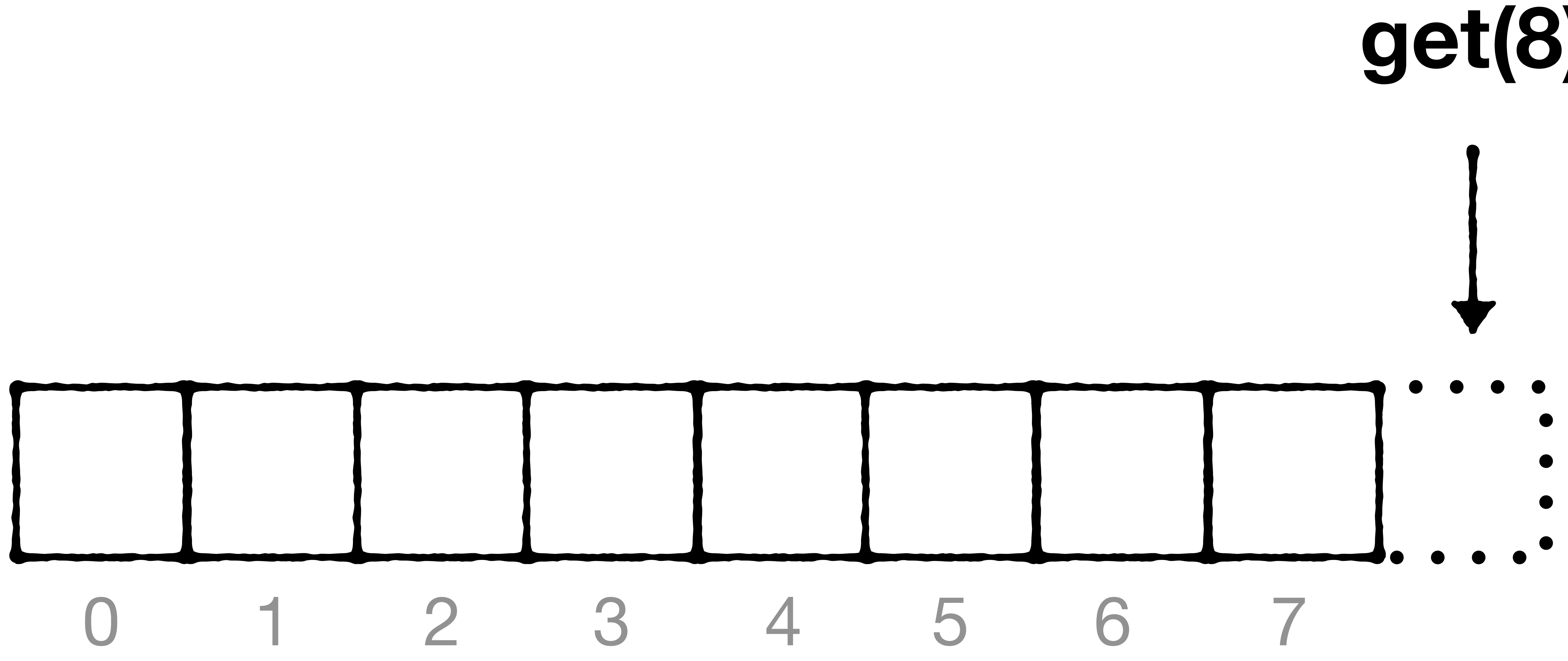
Supports random access



Supports random access



Overflow error (e.g., java)

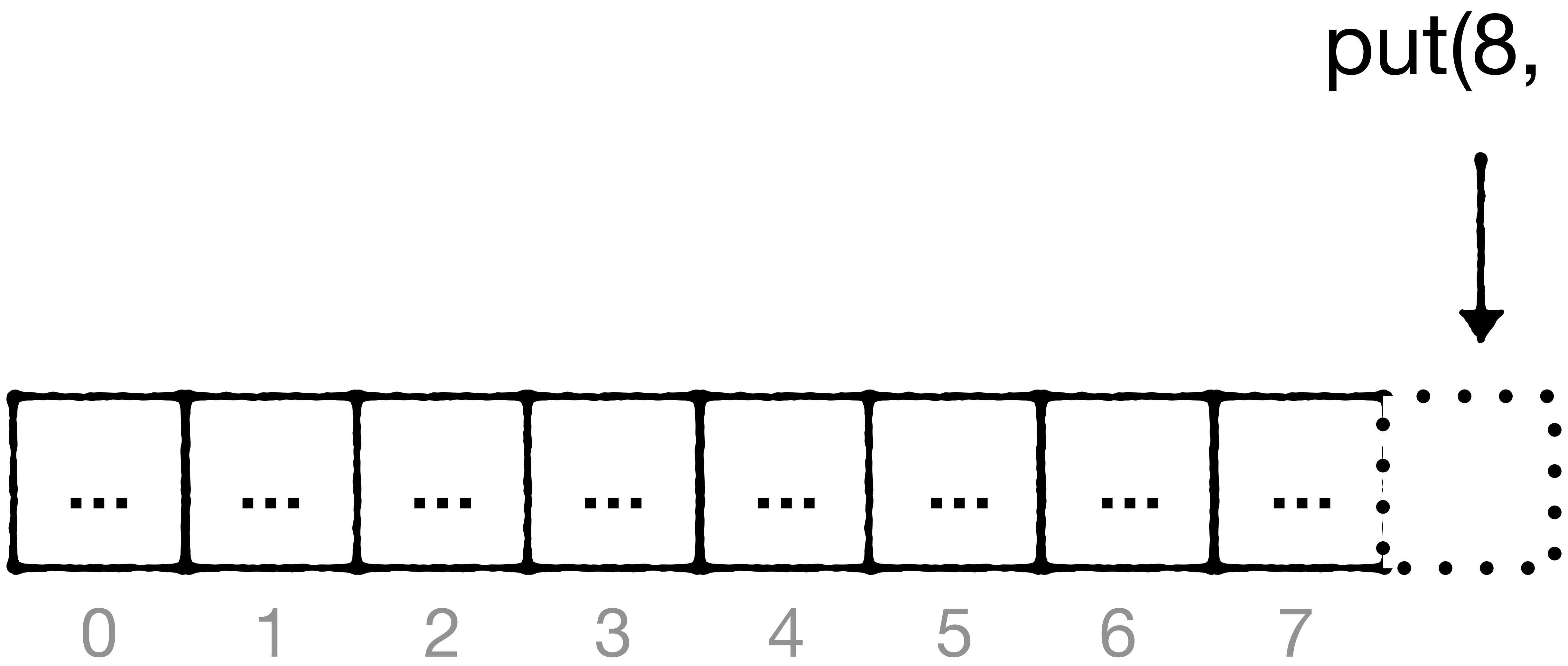


Overflow error (e.g., java)

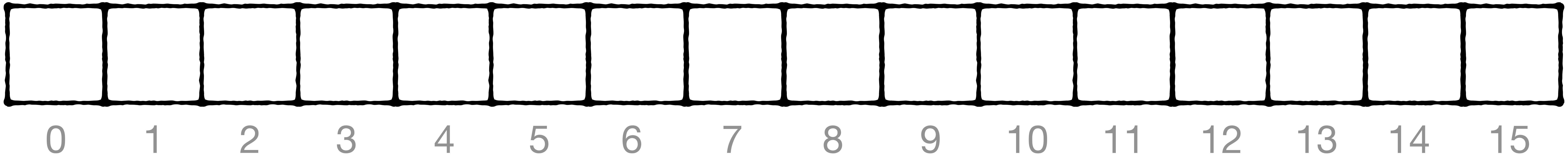
Undefined behavior (e.g., C++)

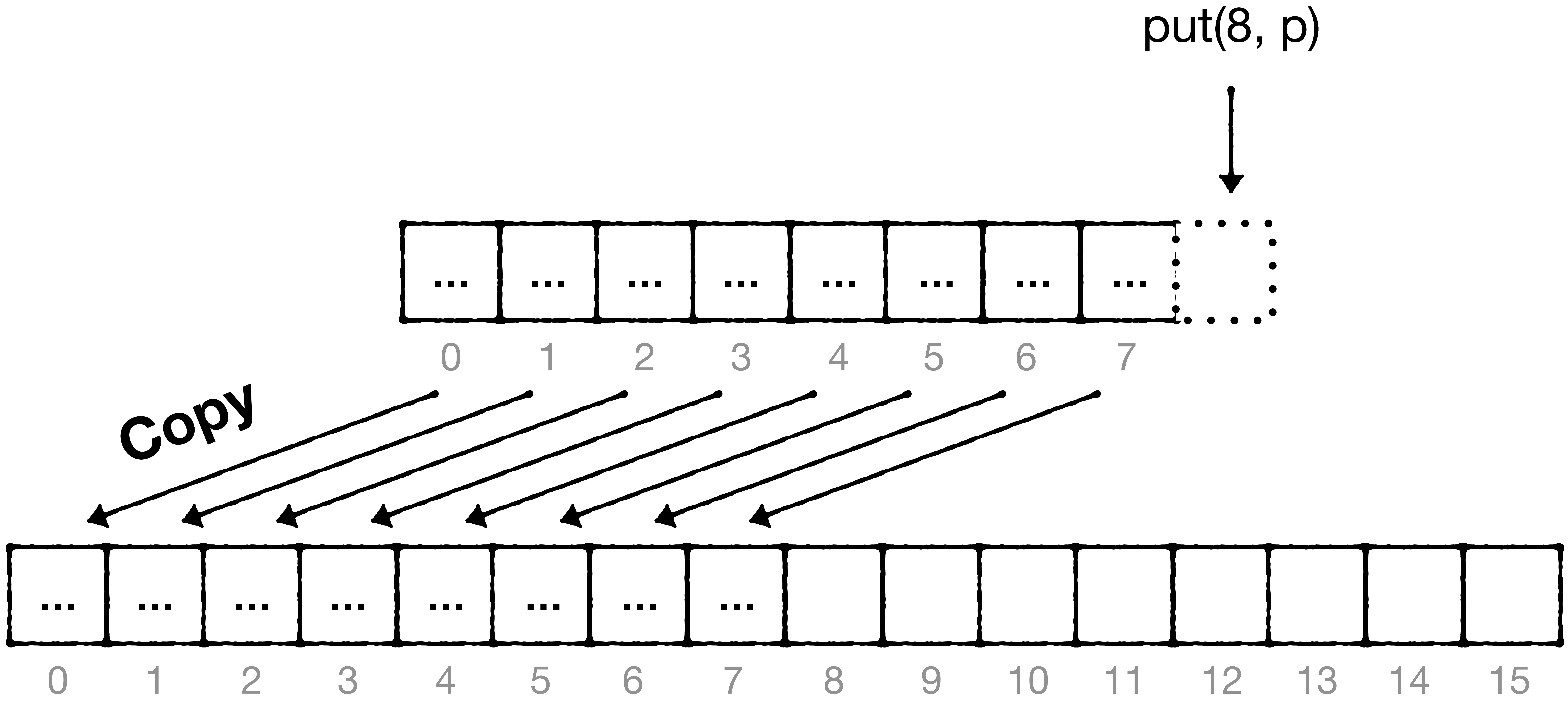
How to keep inserting when out of space?

How to keep inserting when out of space?

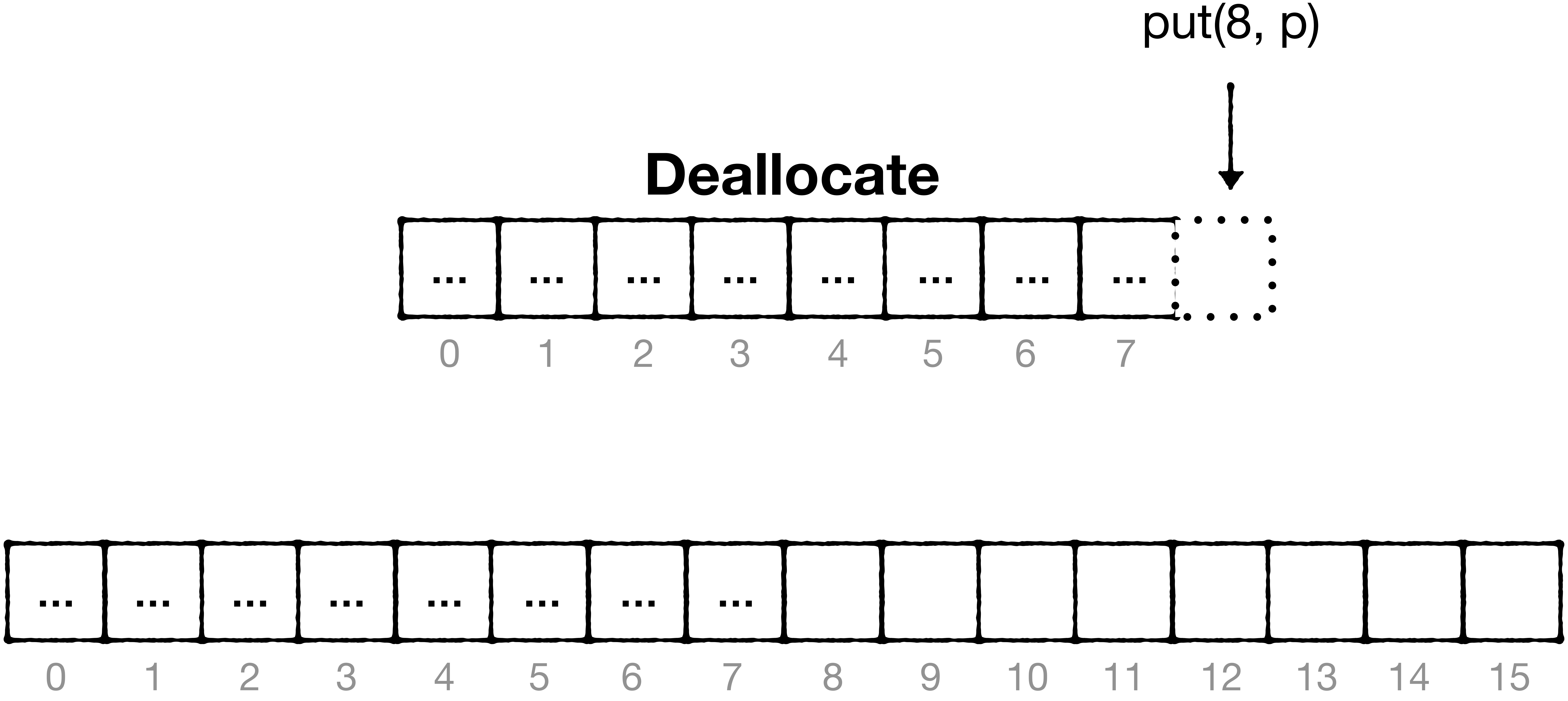


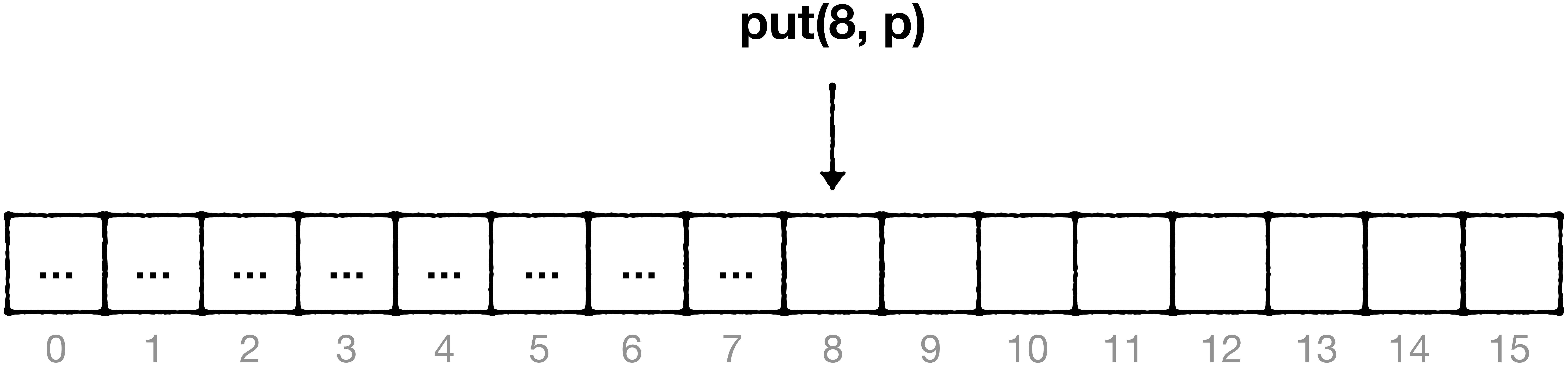
Allocate





put(8, p)
... ...
0 1 2 3 4 5 6 7
Copy
... ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


put(8, p)
Deallocate
... ...
0 1 2 3 4 5 6 7
... ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


put(8, p)
... ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

C++ offers static & dynamic arrays

Static

```
int nums[4] = {0, 0 ,0 ,0};
```

Dynamic

```
std::vector<int> vec;
```

C++ offers static & dynamic arrays

```
std::vector<int> vec;
for (int i = 0; i < 10; ++i) {
    vec.push_back(i);
    std::cout << "Added " << i << ", size: " << vec.size()
            << ", capacity: " << vec.capacity() << std:end;
}
```

C++ offers static & dynamic arrays

```
std::vector<int> vec;
for (int i = 0; i < 10; ++i) {
    vec.push_back(i);
    std::cout << "Added " << i << ", size: " << vec.size()
            << ", capacity: " << vec.capacity() << std:end;
}
```

Resize if we exceed capacity

C++ offers static & dynamic arrays

```
std::vector<int> vec;
for (int i = 0; i < 10; ++i) {
    vec.push_back(i);
    std::cout << "Added " << i << ", size: " << vec.size()
            << ", capacity: " << vec.capacity() << std:end;
}
```

Element	Size	Capacity
0	1	1
1	2	2
2	3	4
3	4	4
4	5	8
5	6	8
6	7	8
7	8	8
8	9	16
9	10	16

C++ offers static & dynamic arrays

```
std::vector<int> vec;
for (int i = 0; i < 10; ++i) {
    vec.push_back(i);
    std::cout << "Added " << i << ", size: " << vec.size()
            << ", capacity: " << vec.capacity() << std:end;
}
```

Element	Size	**Capacity**
0	1	**1**
1	2	**2**
2	3	**4**
3	4	**4**
4	5	**8**
5	6	**8**
6	7	**8**
7	8	**8**
8	9	**16**
9	10	**16**

Uses Growth Factor 2

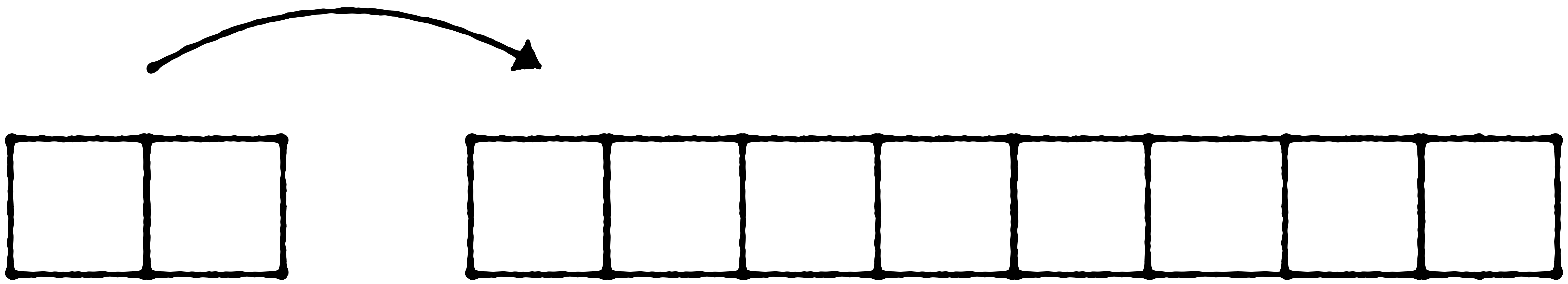
What if Growth Factor G is

too high?

too low?

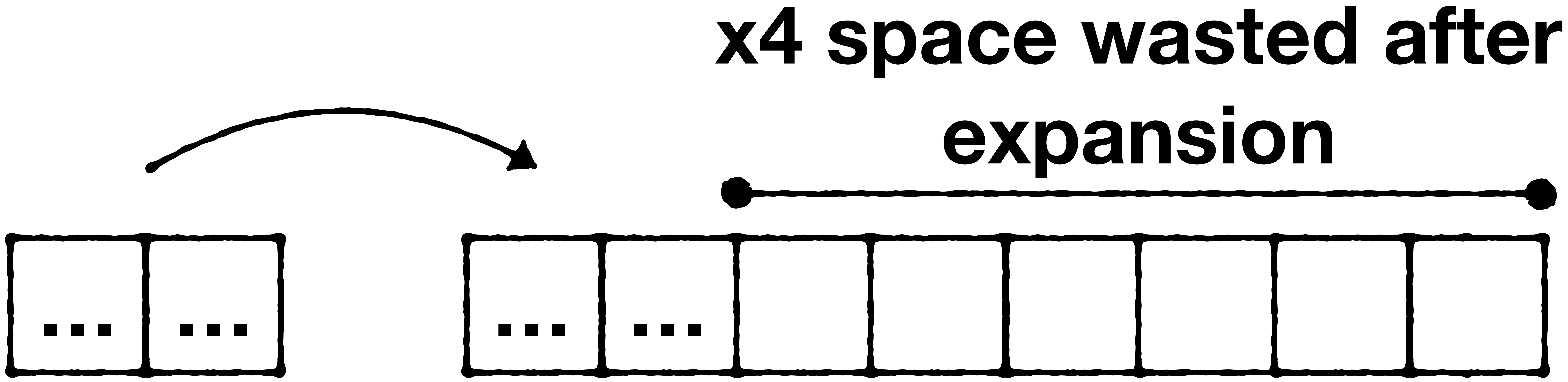
What if Growth Factor G is **too high**

e.g., 4



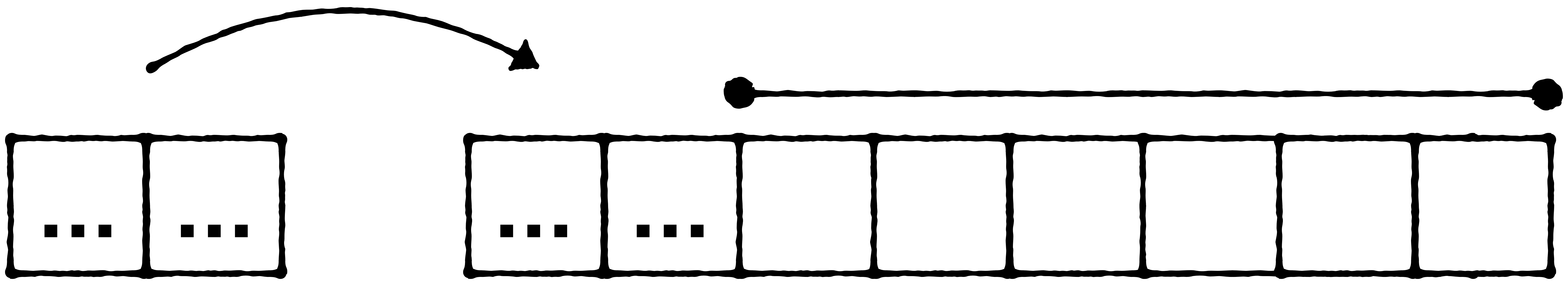
What if Growth Factor G is **too high**

e.g., 4



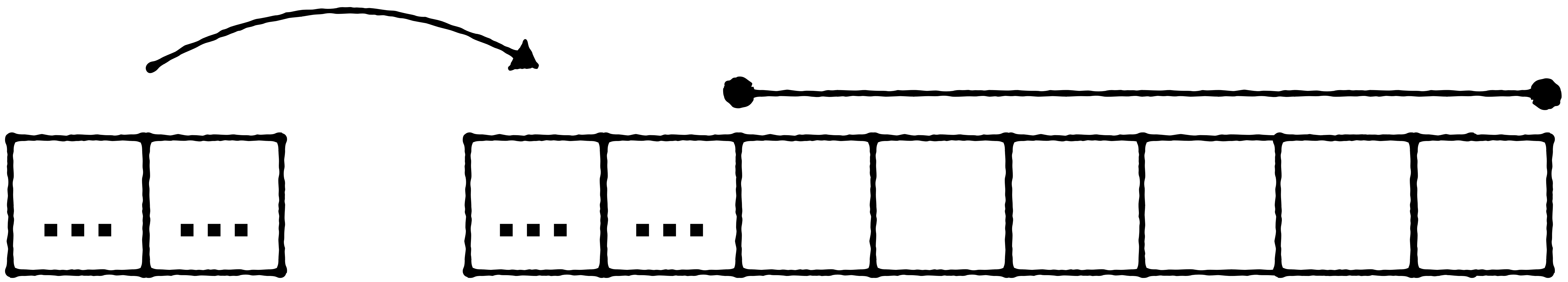
What if Growth Factor G is **too high**

$$\textbf{Space-amplification} = \frac{\textbf{Physical space used}}{\textbf{Data size}}$$



What if Growth Factor G is **too high**

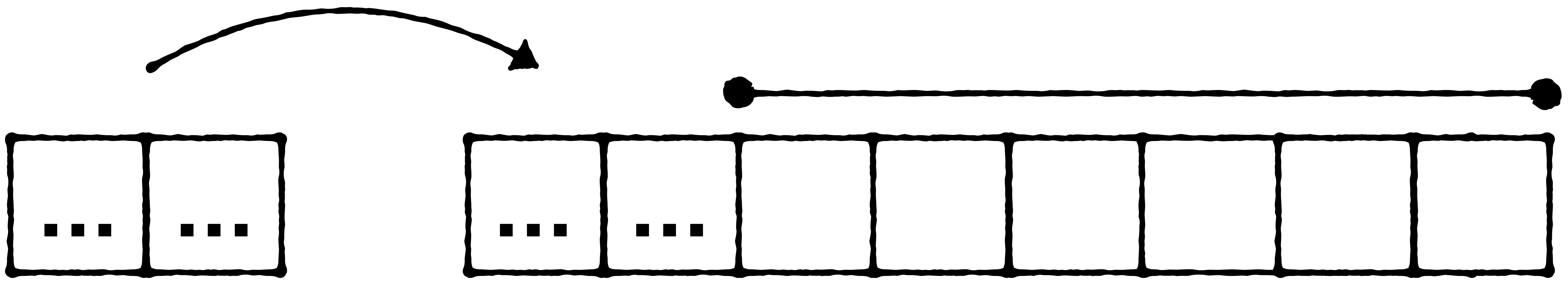
$$\textbf{Space-amplification} = \frac{\textbf{Physical space used}}{\textbf{Data size}} = \textbf{G}$$



What if Growth Factor G is **too high**

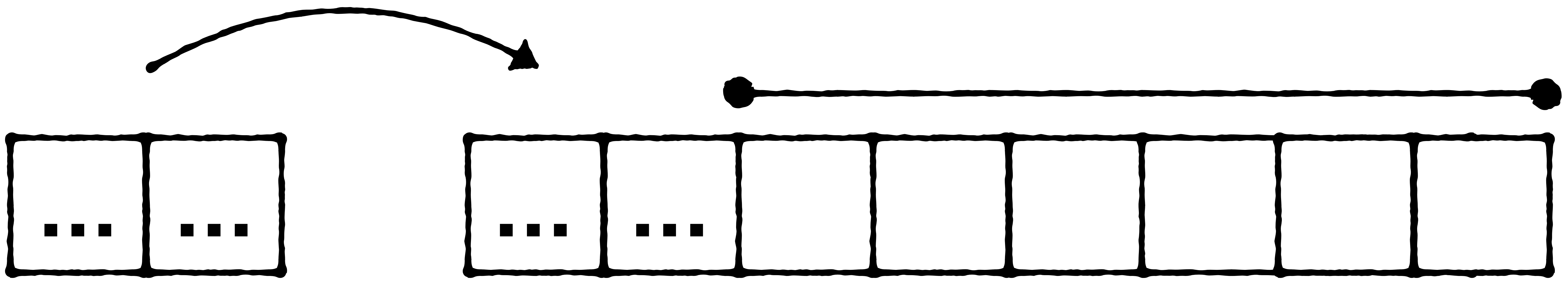
Space-amplification = G

Right after expansion



What if Growth Factor G is **too high**

Max Space-amplification = **G + 1** **During expansion**

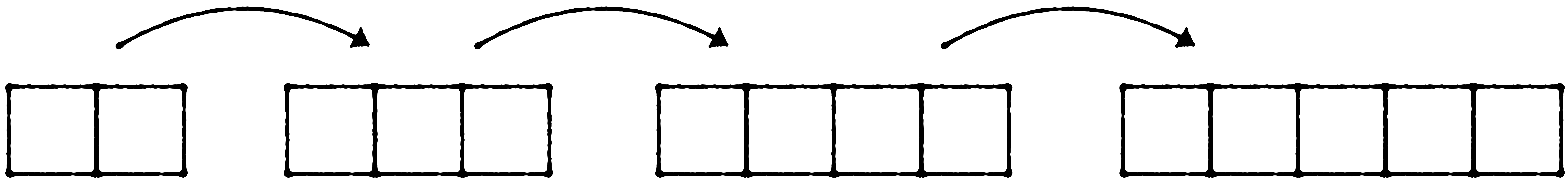


What if Growth Factor G is **too low**

e.g., 1.2

What if Growth Factor G is **too low**

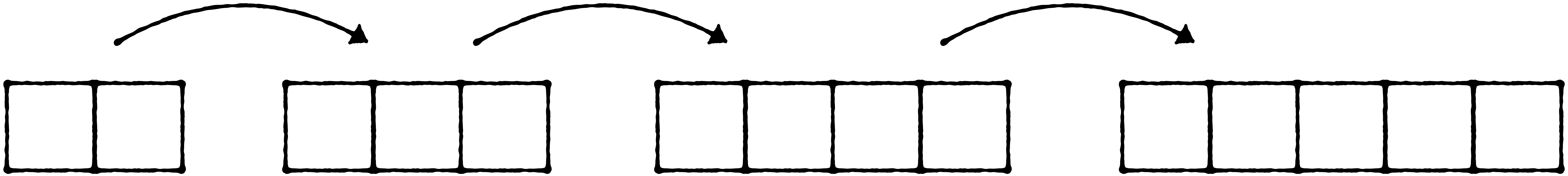
e.g., 1.2



What if Growth Factor G is **too low**

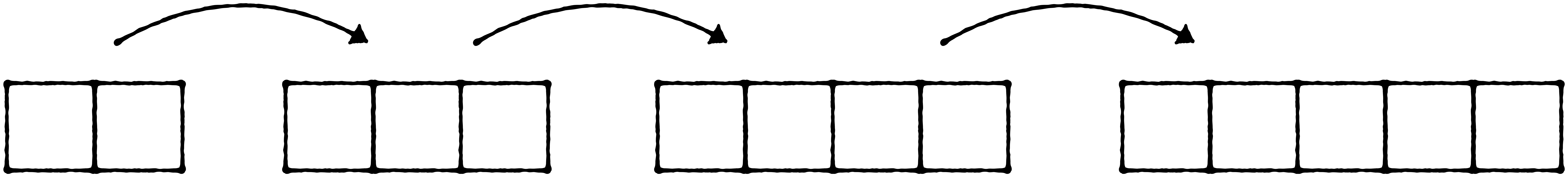
e.g., 1.2

Insertion overheads increase



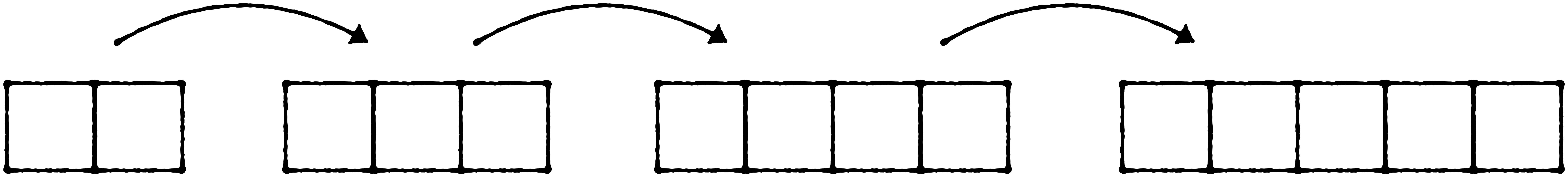
What if Growth Factor G is **too low**

$$\textbf{Write-amplification} = \frac{\textbf{Physical data written}}{\textbf{Data size}}$$



What if Growth Factor G is **too low**

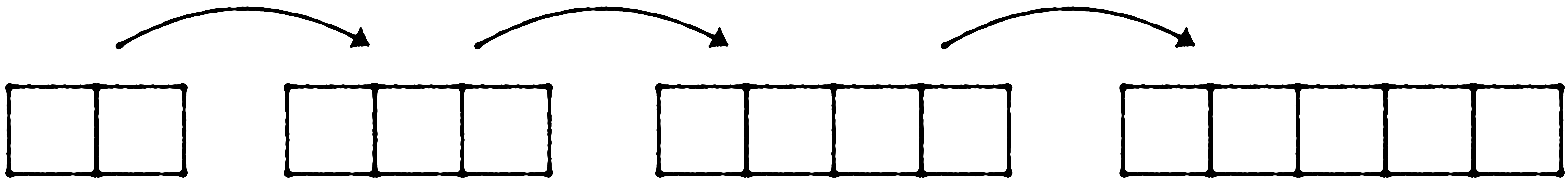
$$\textbf{Write-amplification} = \frac{\textbf{Physical data written}}{\textbf{Data size}} = \frac{G}{G - 1}$$



What if Growth Factor G is **too low**

$$\text{Write-amplification} = \frac{\text{Physical data written}}{\text{Data size}} = \frac{G}{G - 1}$$

Geometric series sum



Growth factor G impact

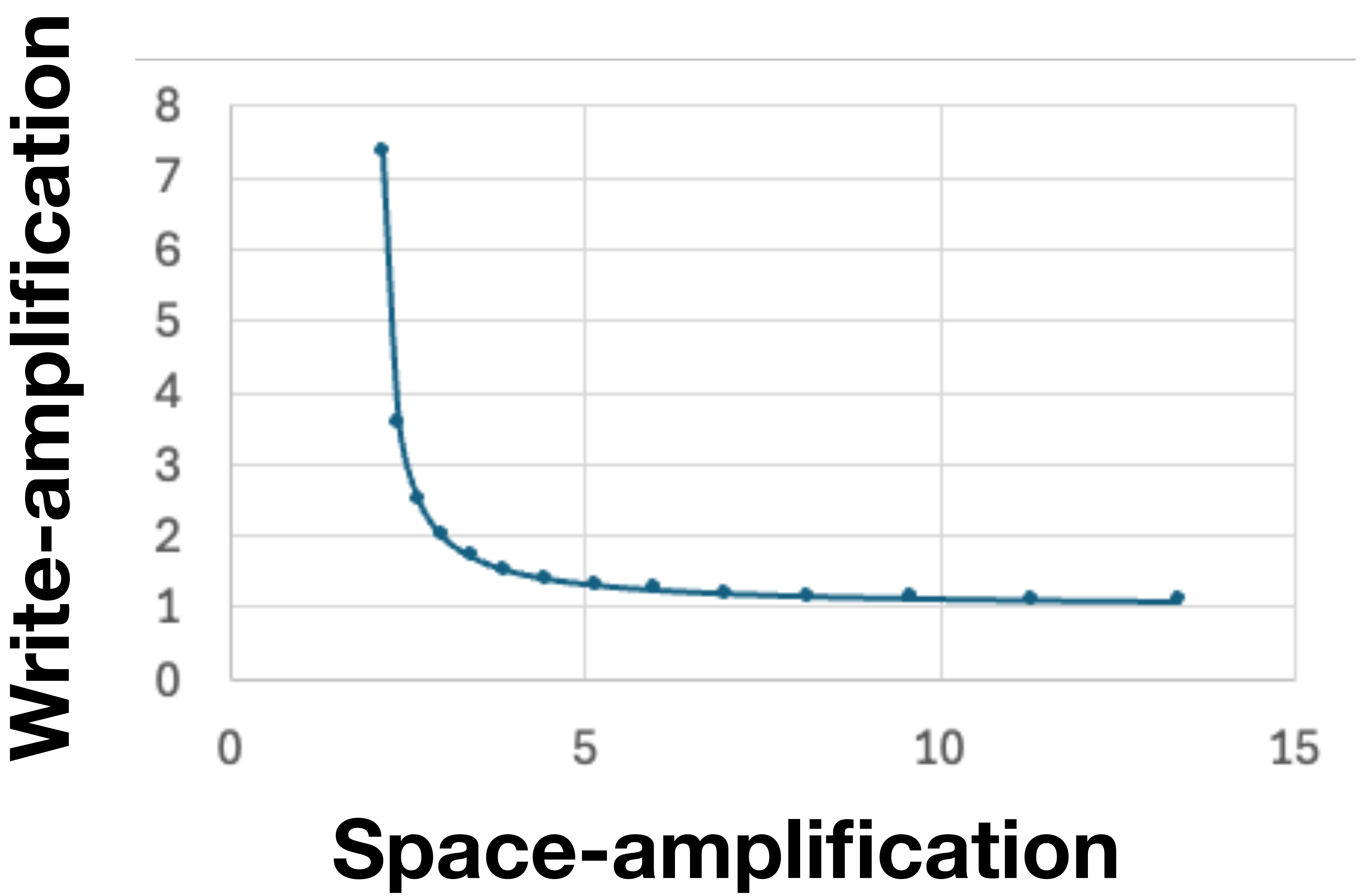
Space-amplification

$$G+1$$

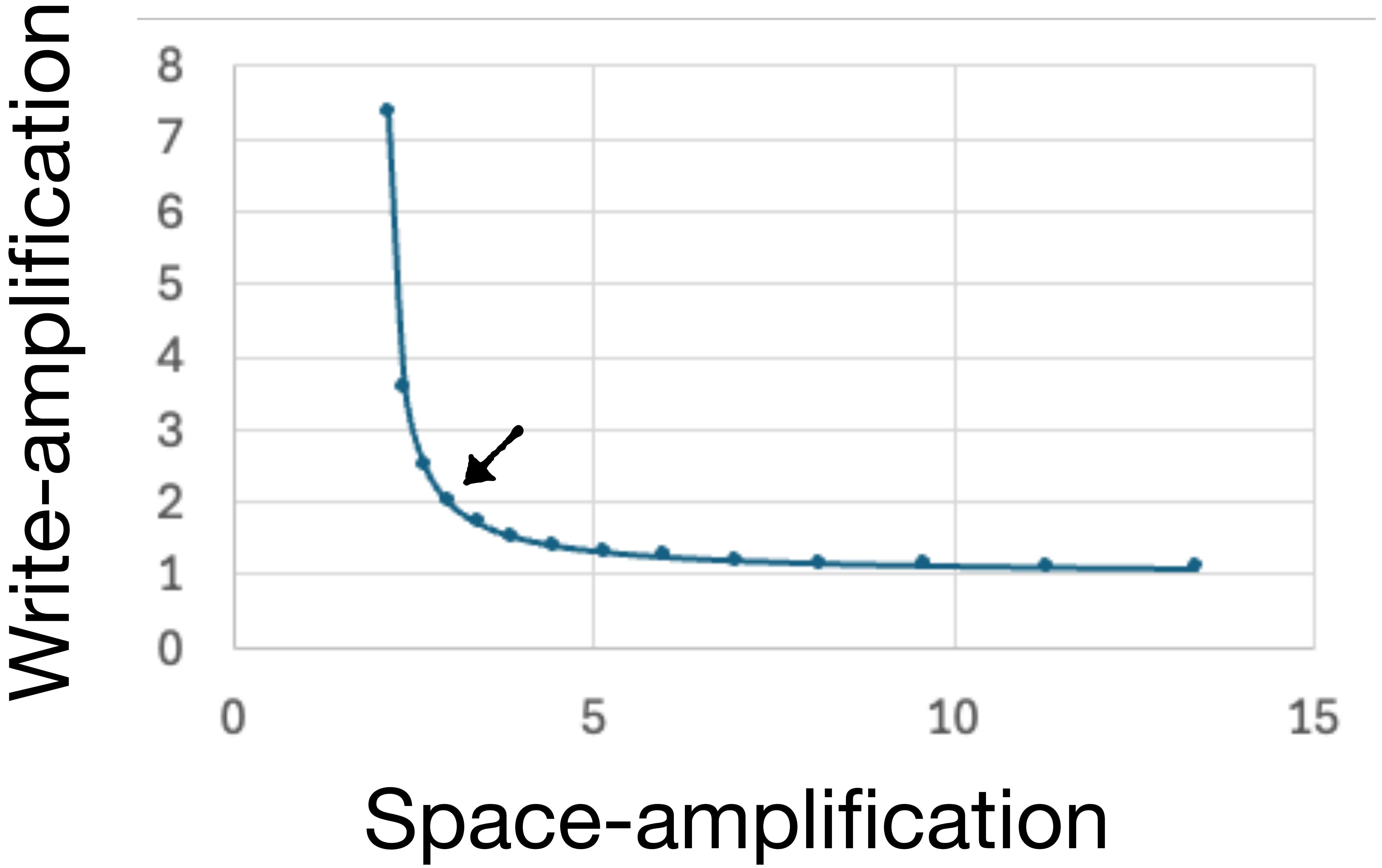
Write-amplification

$$\frac{G}{G - 1}$$

Growth factor G impact

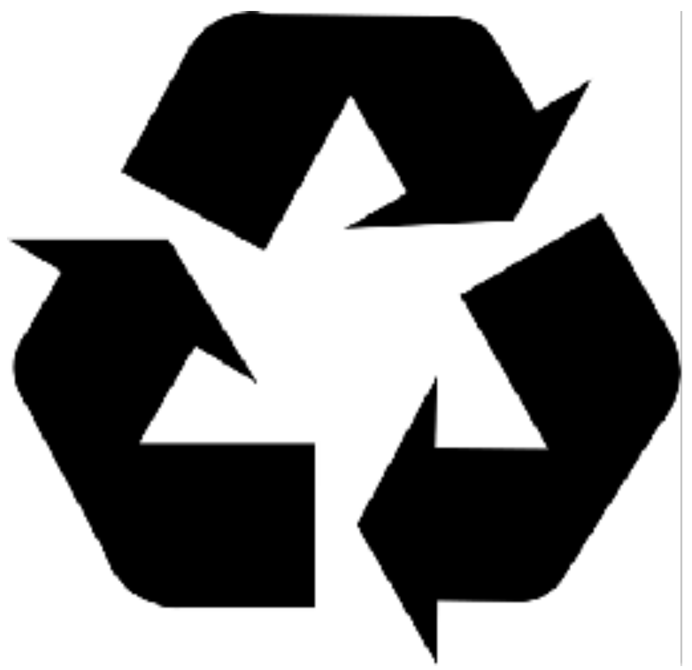


Growth factor 2 achieves a good balance

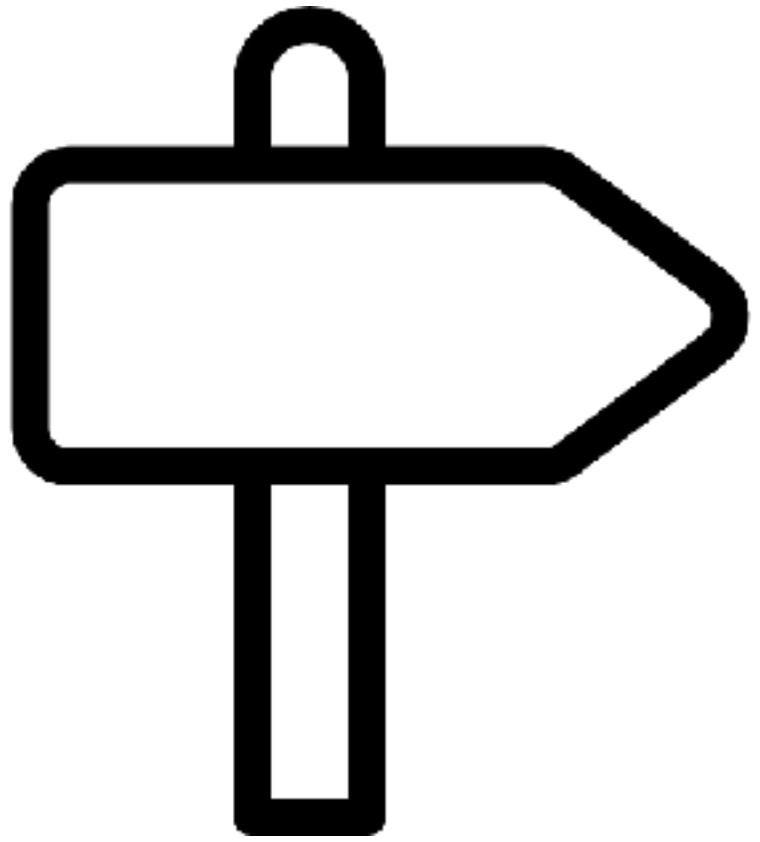


And now to new stuff

Reusing Deallocated Space

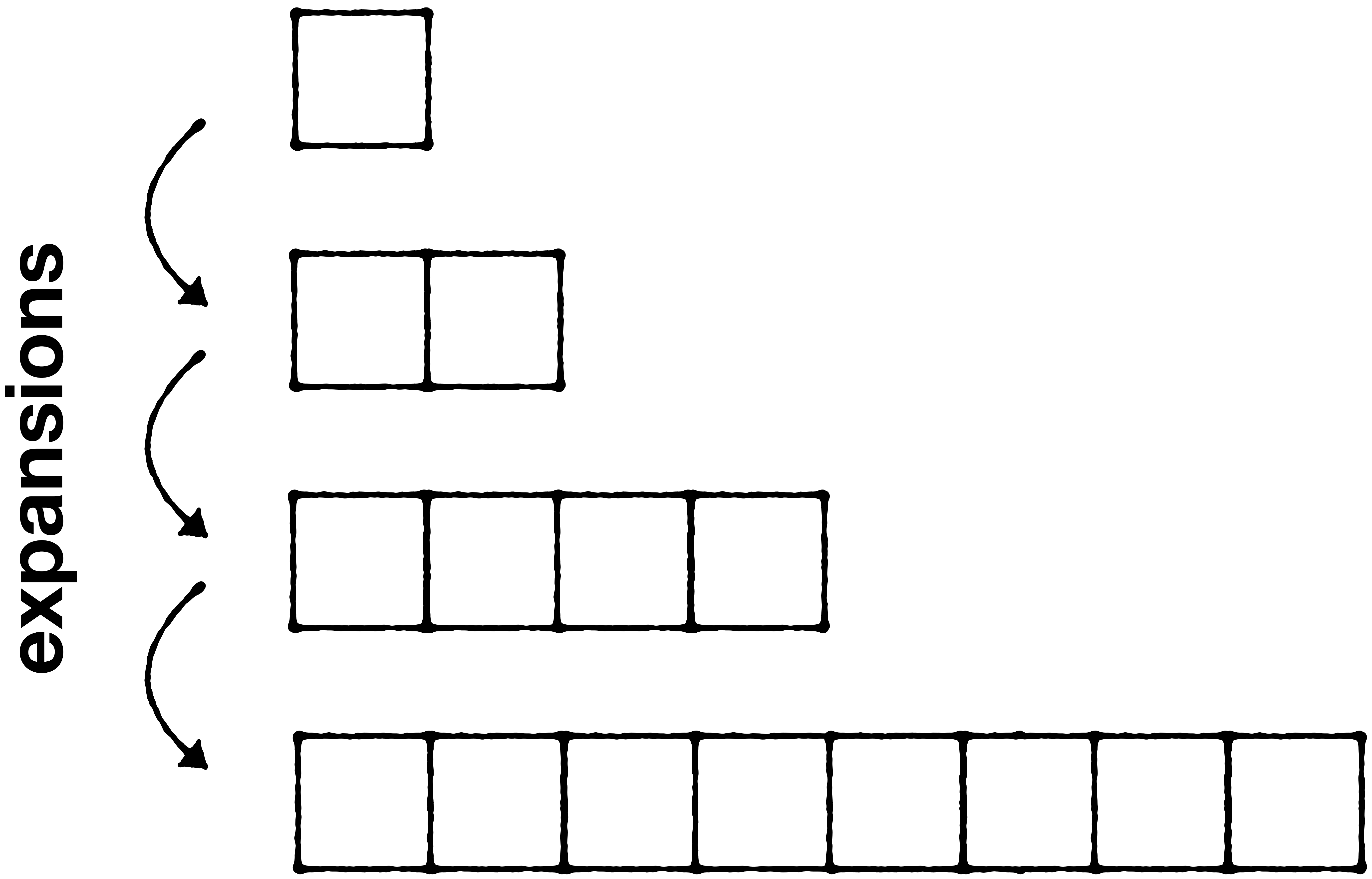


Alleviating trade-off via indirection

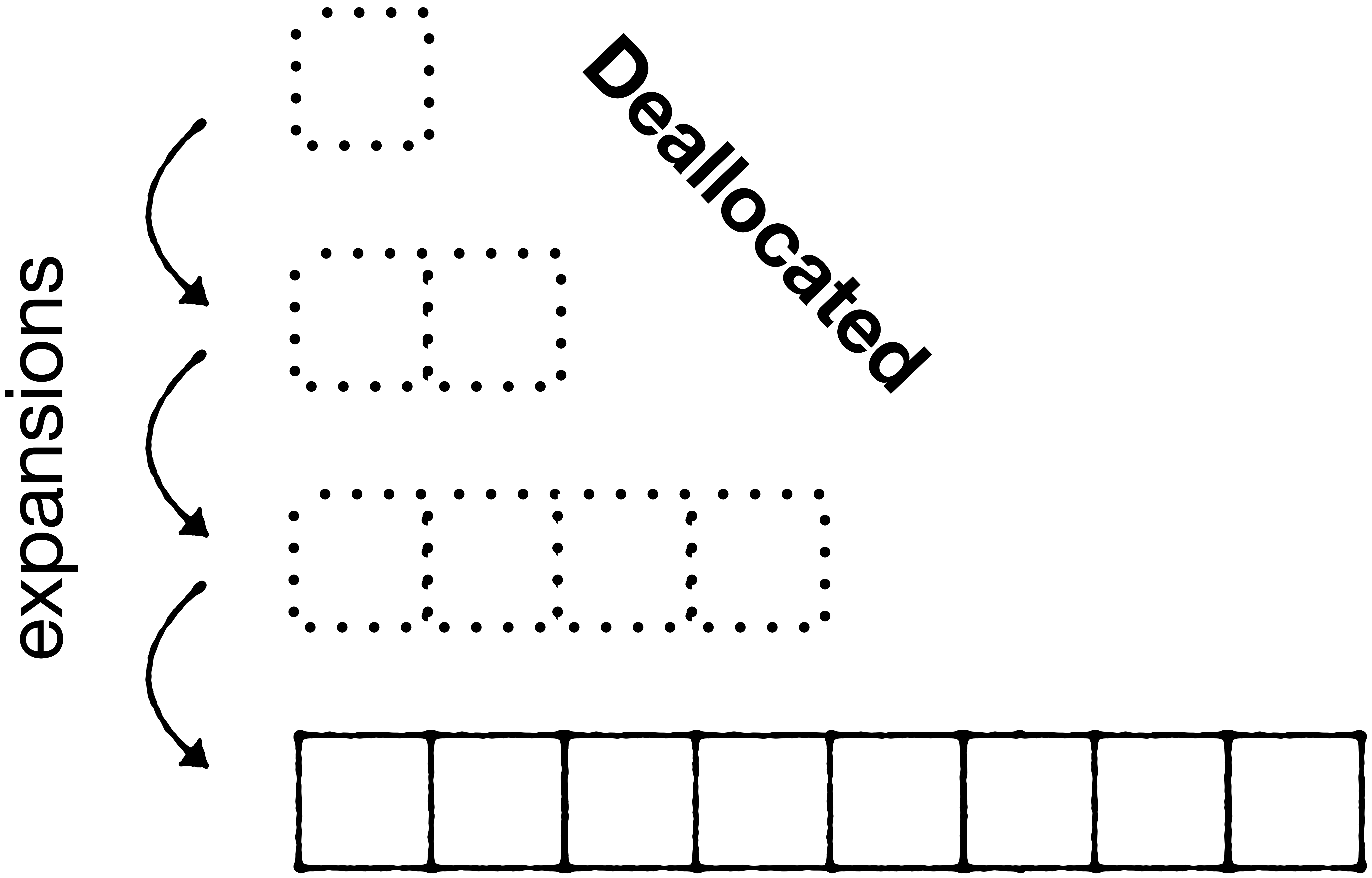


Reusing Deallocated Space

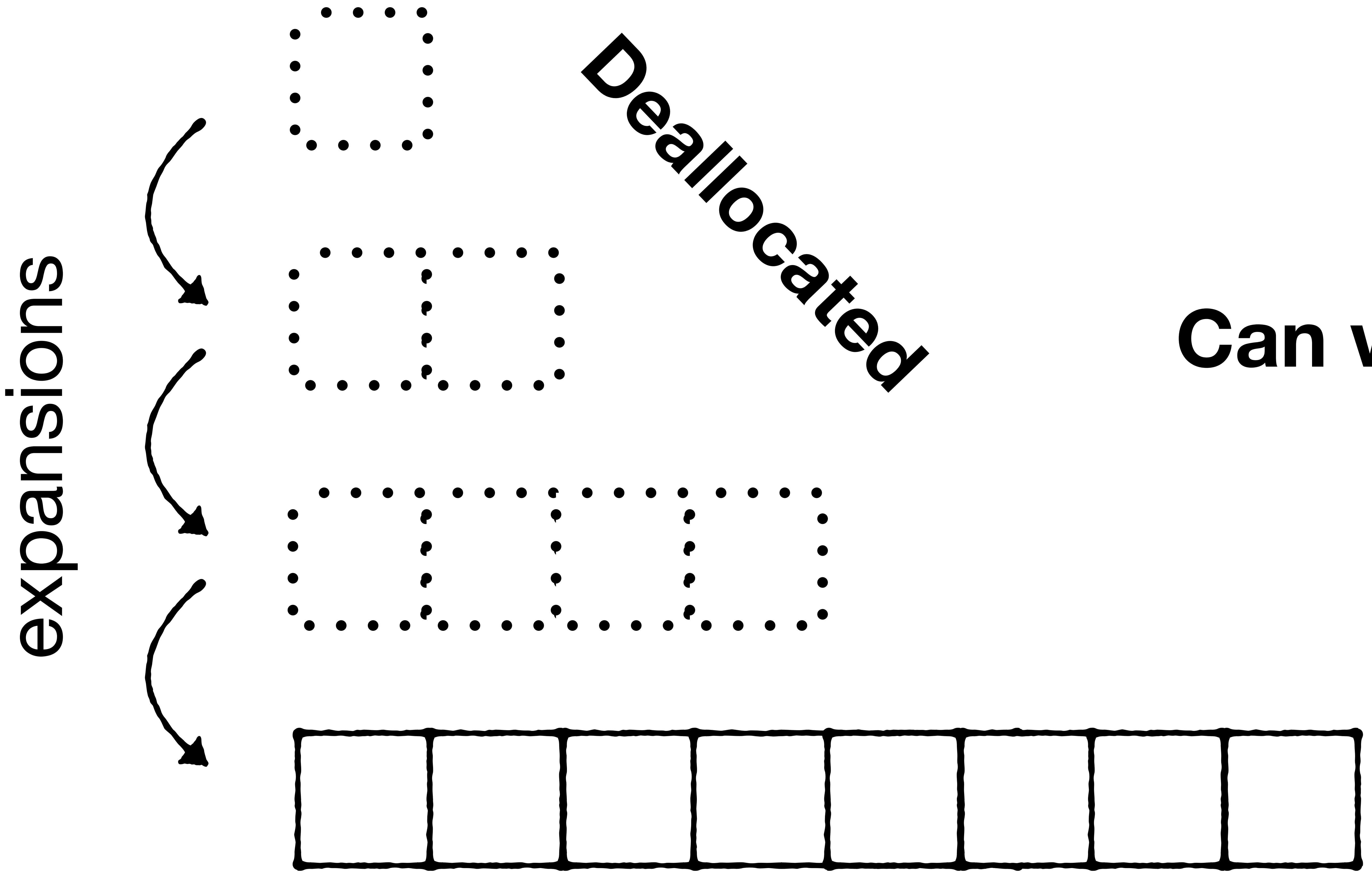
Assume G ≥ 2



Reusing Deallocated Space (G ≥ 2)

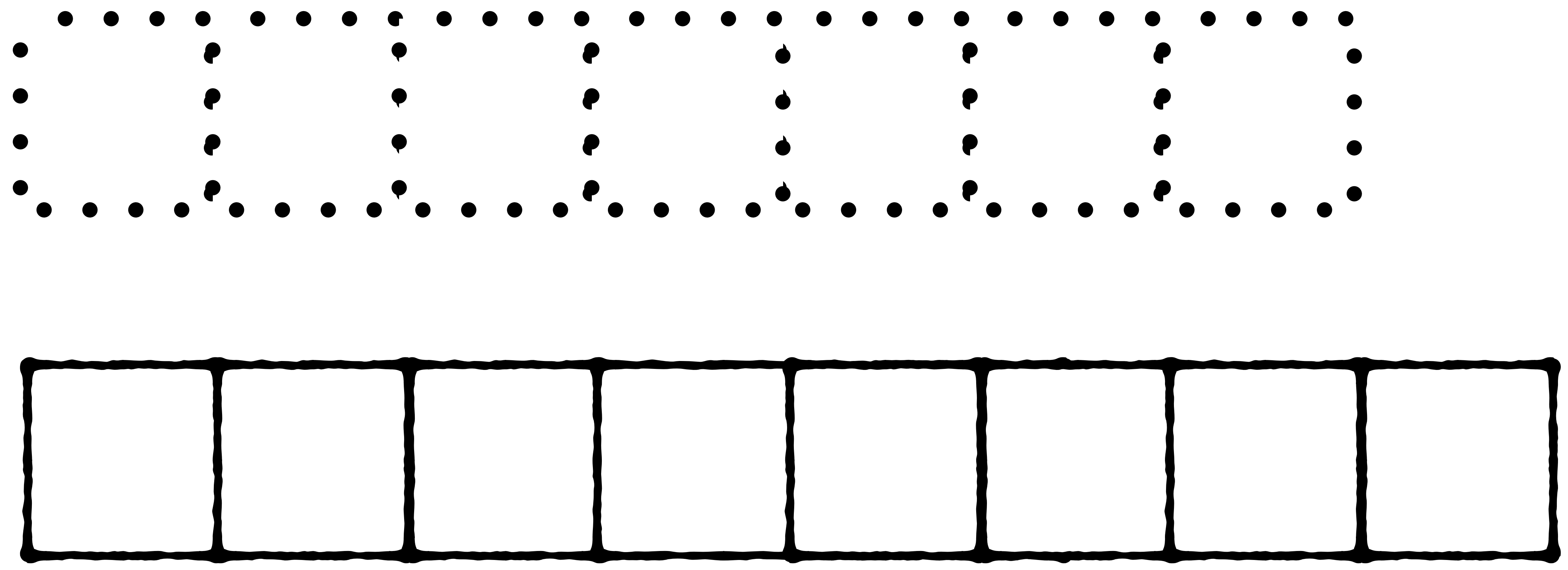


Reusing Deallocated Space (G ≥ 2)



Can we reuse this space?

Reusing Deallocated Space (G ≥ 2)

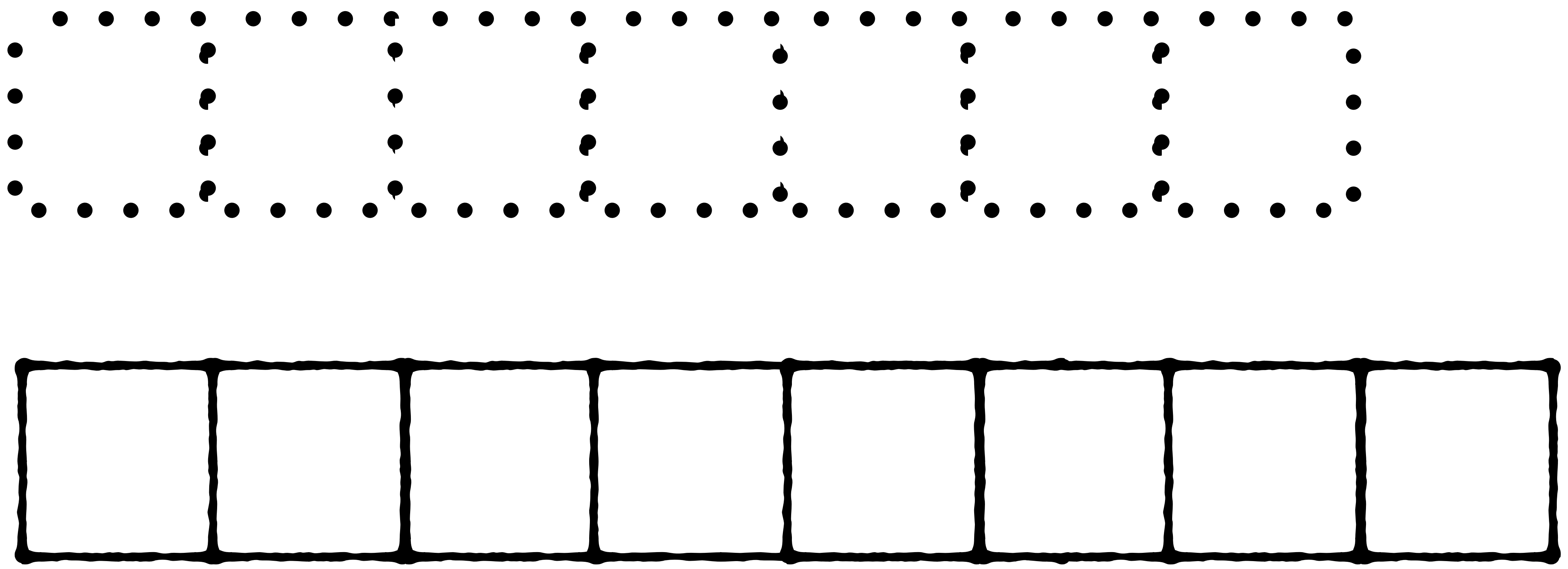


Total deallocated space

=

Total allocated space - 1

Reusing Deallocated Space ($G \geq 2$)

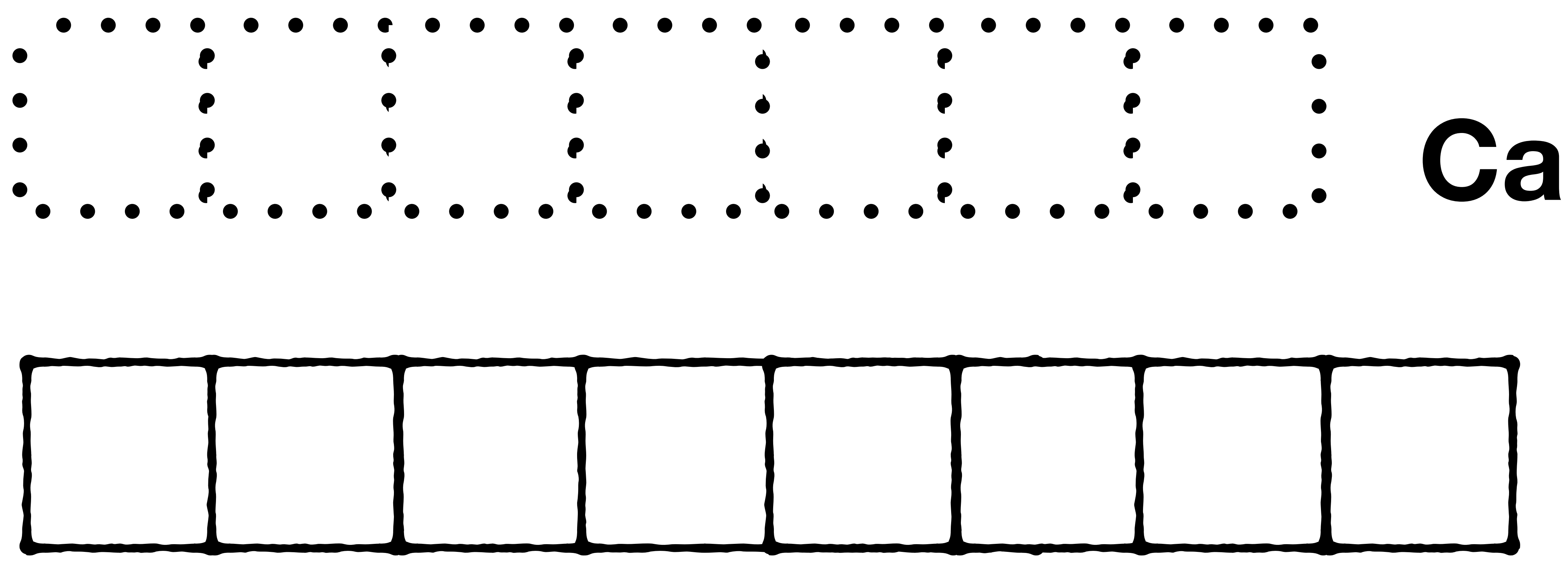


Total deallocated space

∧

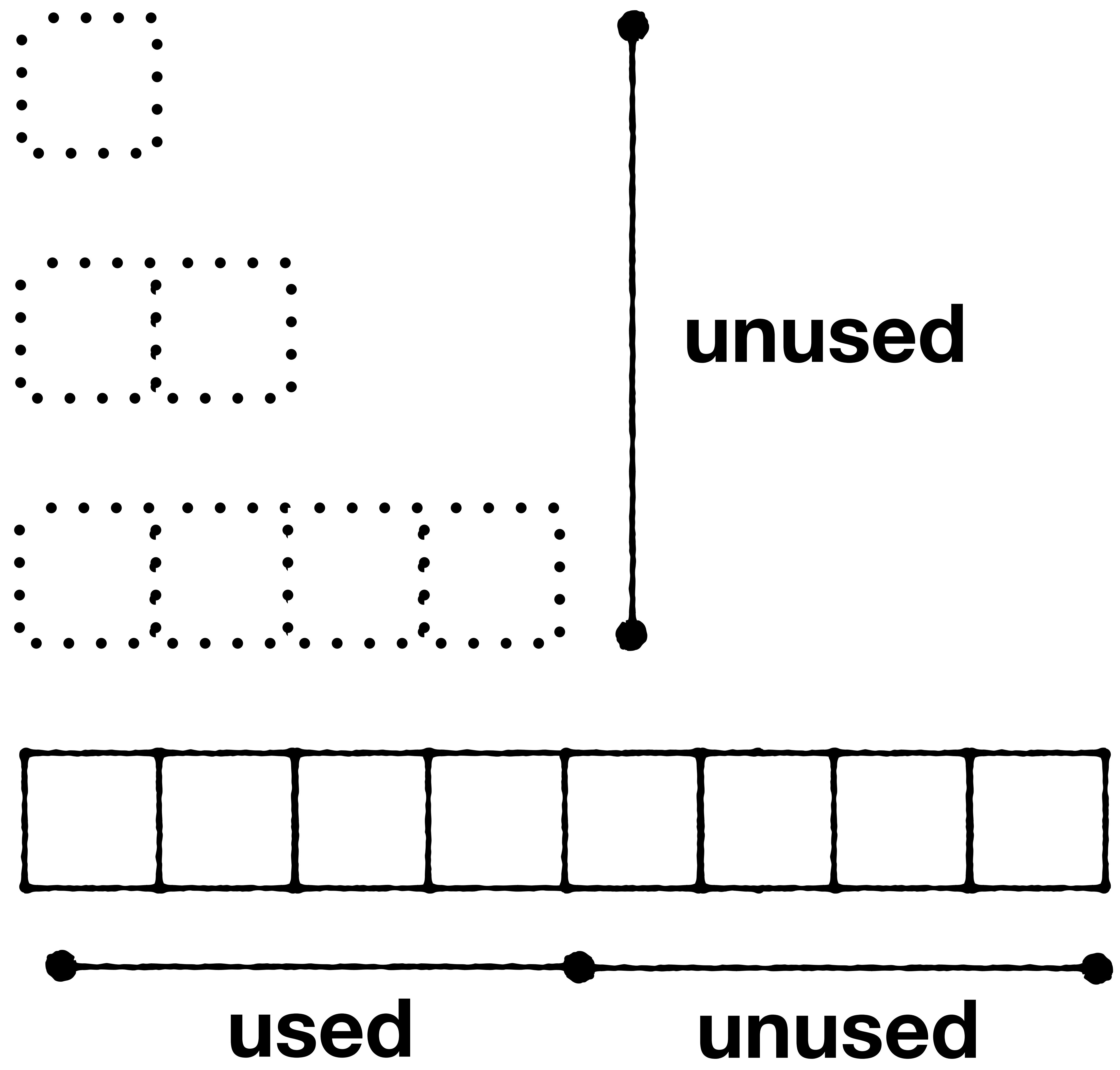
Total allocated space

Reusing Deallocated Space (G ≥ 2)

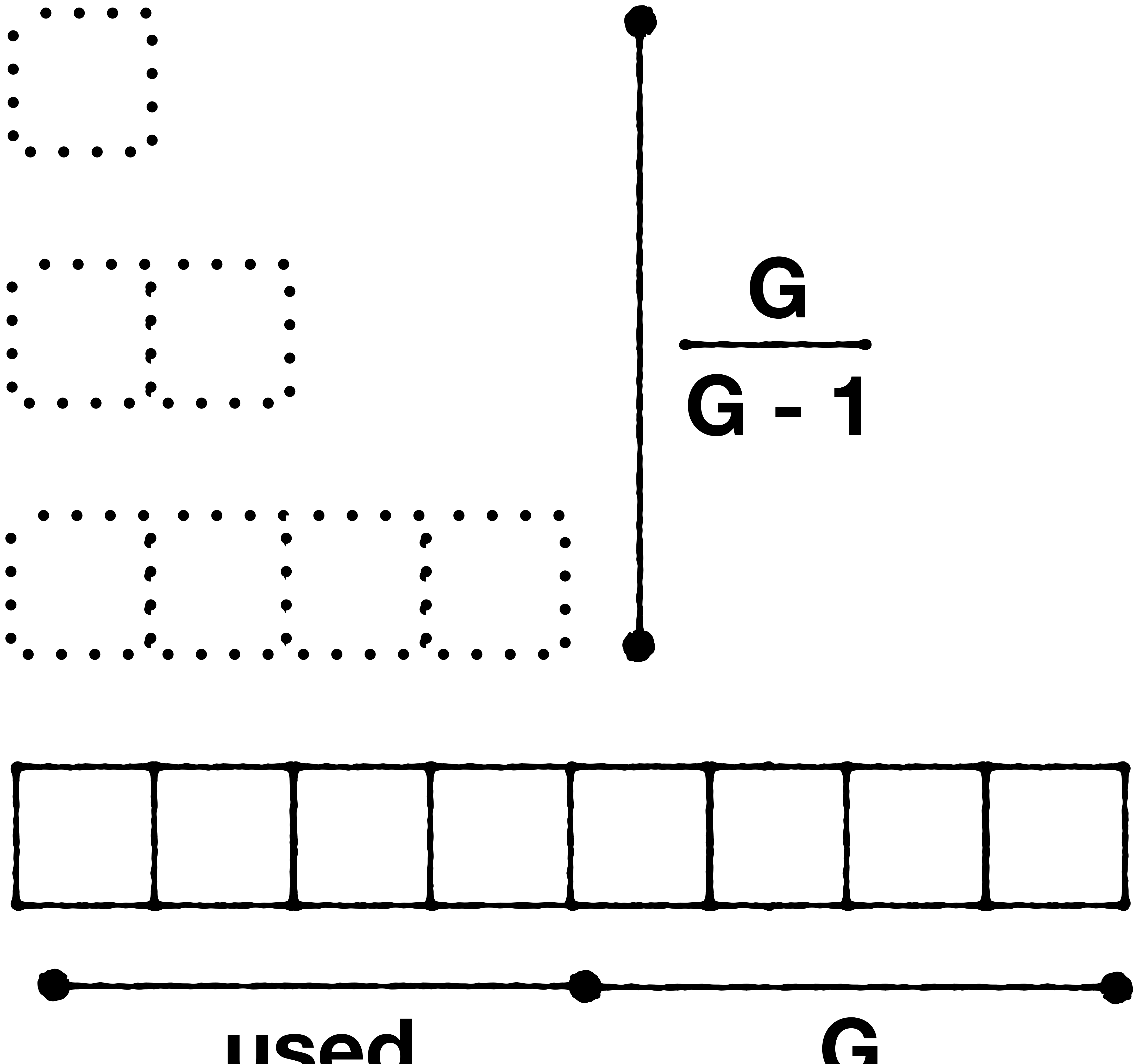


Deallocated space
Can’t be reused by new array

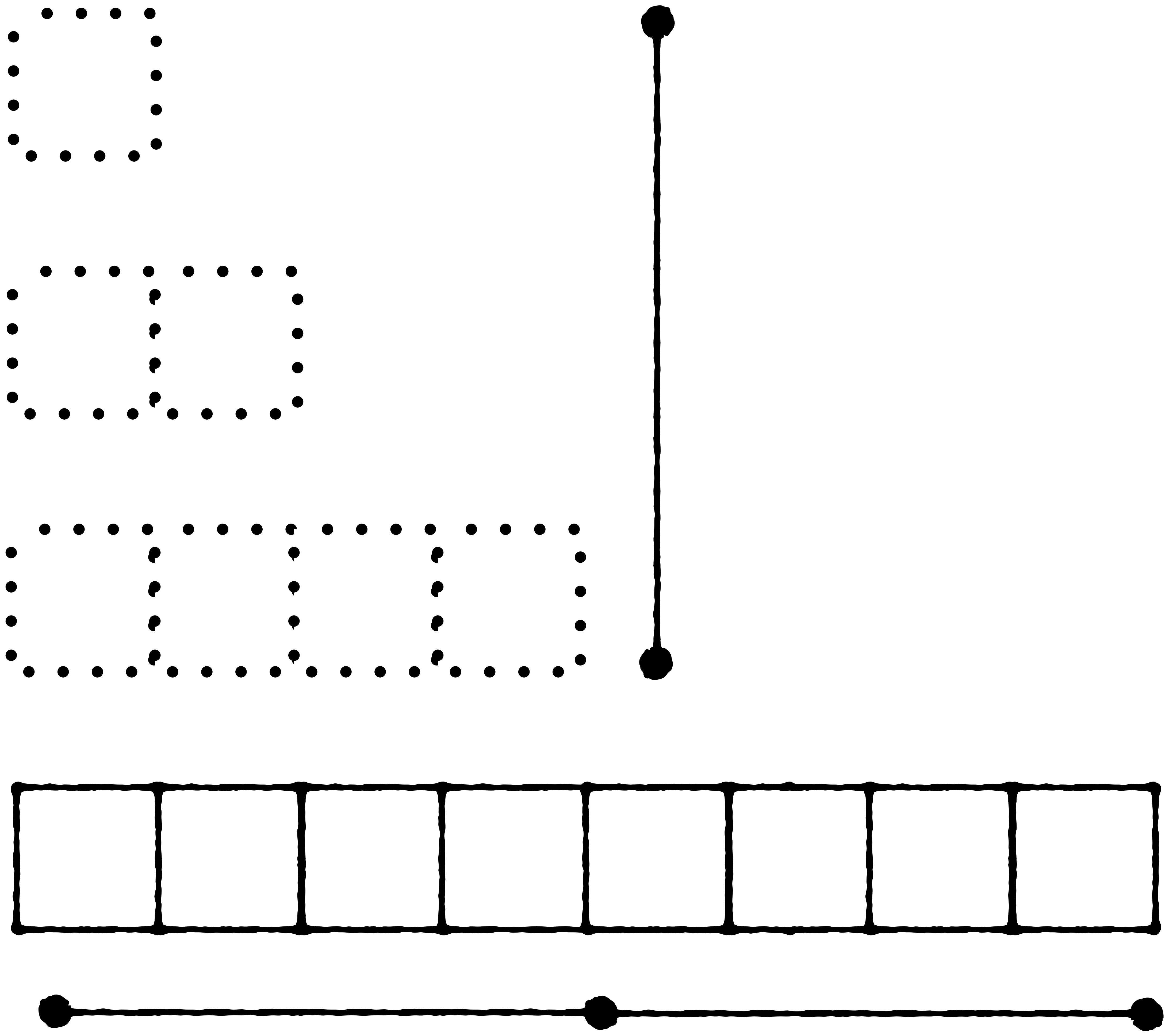
Deriving Max Space-Amp



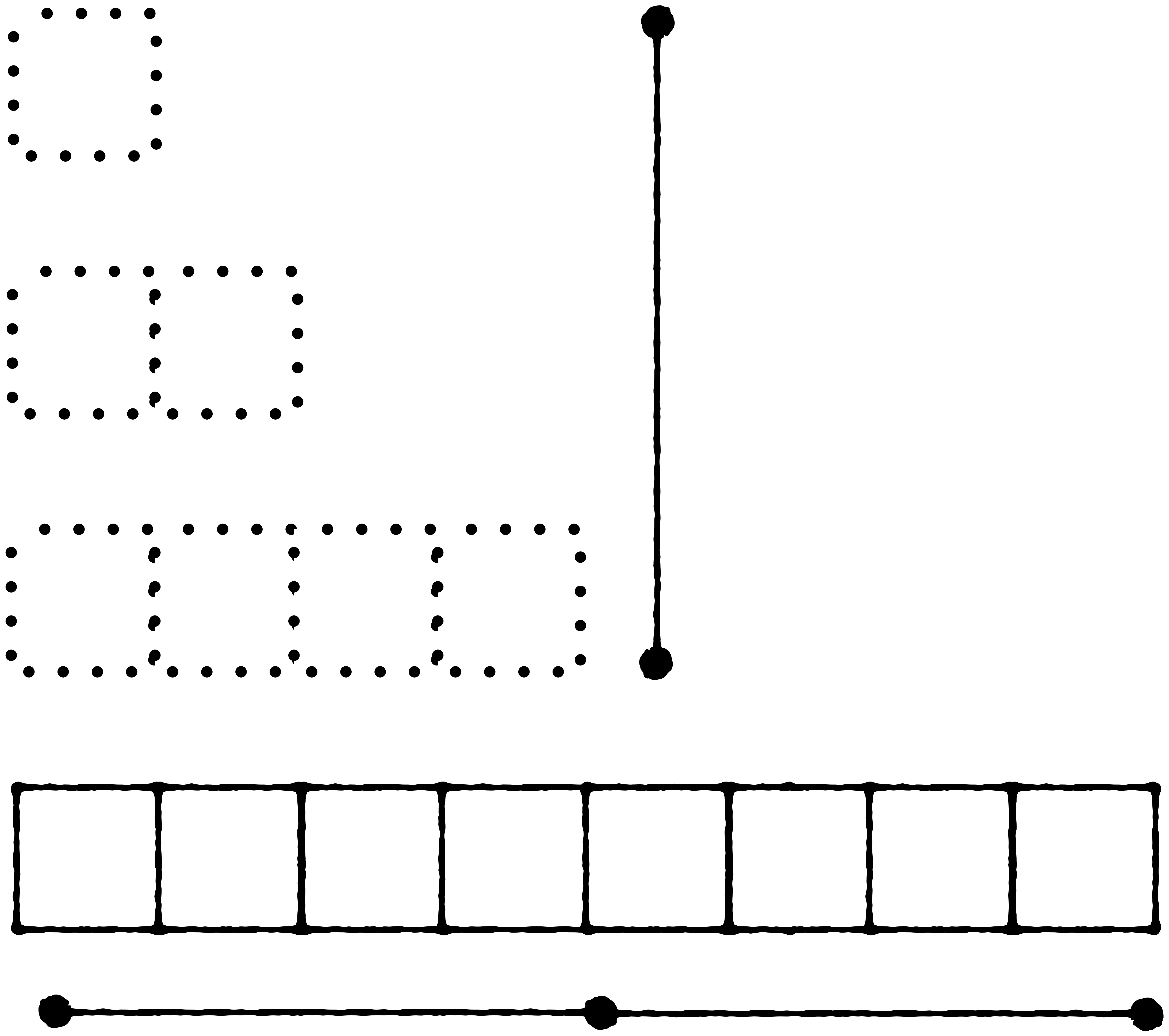
Deriving Max Space-Amp



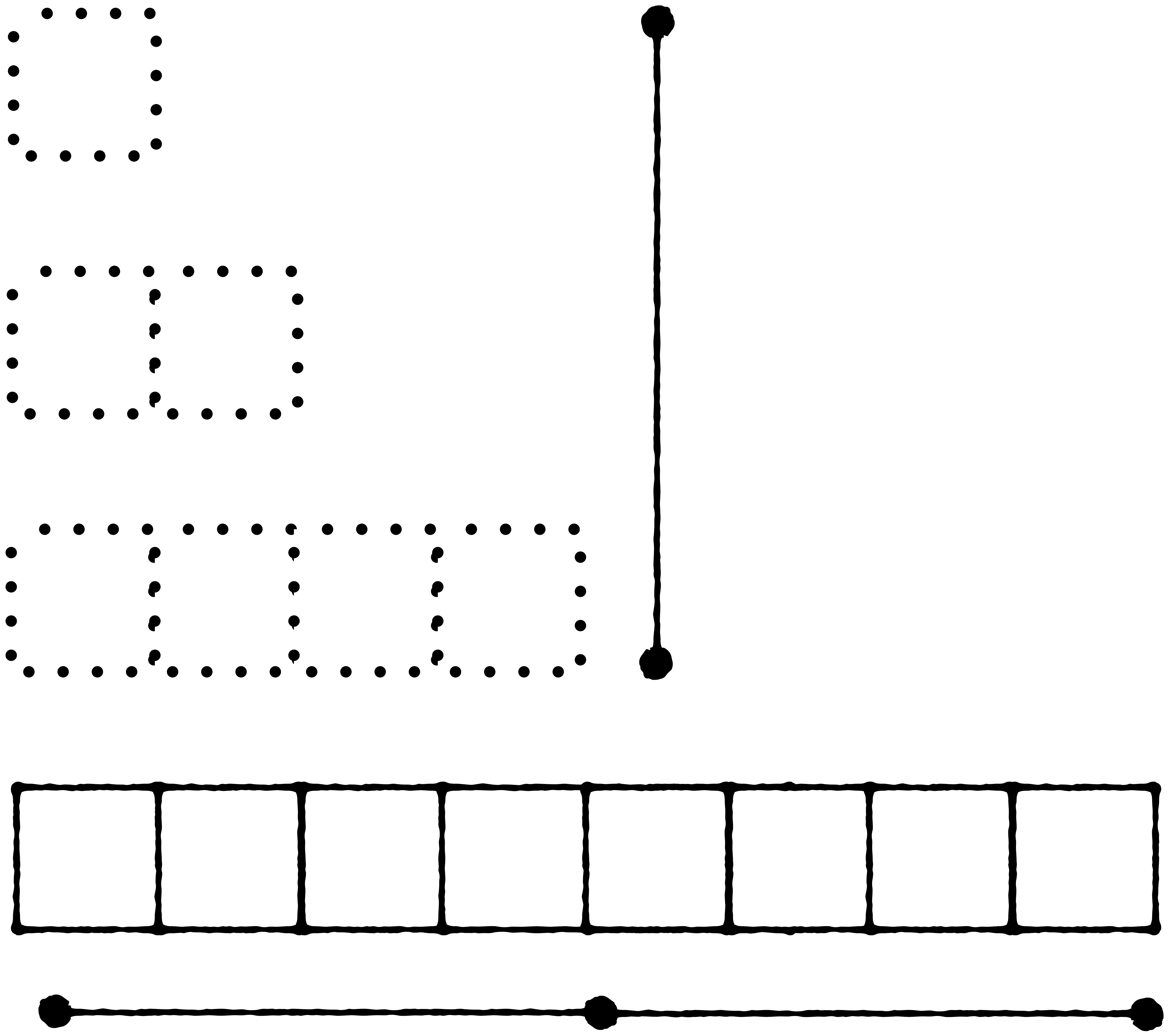
$$\text{Max Space-Amp} = \frac{\text{used + unused}}{\text{used}} = \frac{G}{G - 1} + G$$



$$\text{Max Space-Amp} = \frac{\text{used} + \text{unused}}{\text{used}} = \frac{G^2}{G - 1}$$

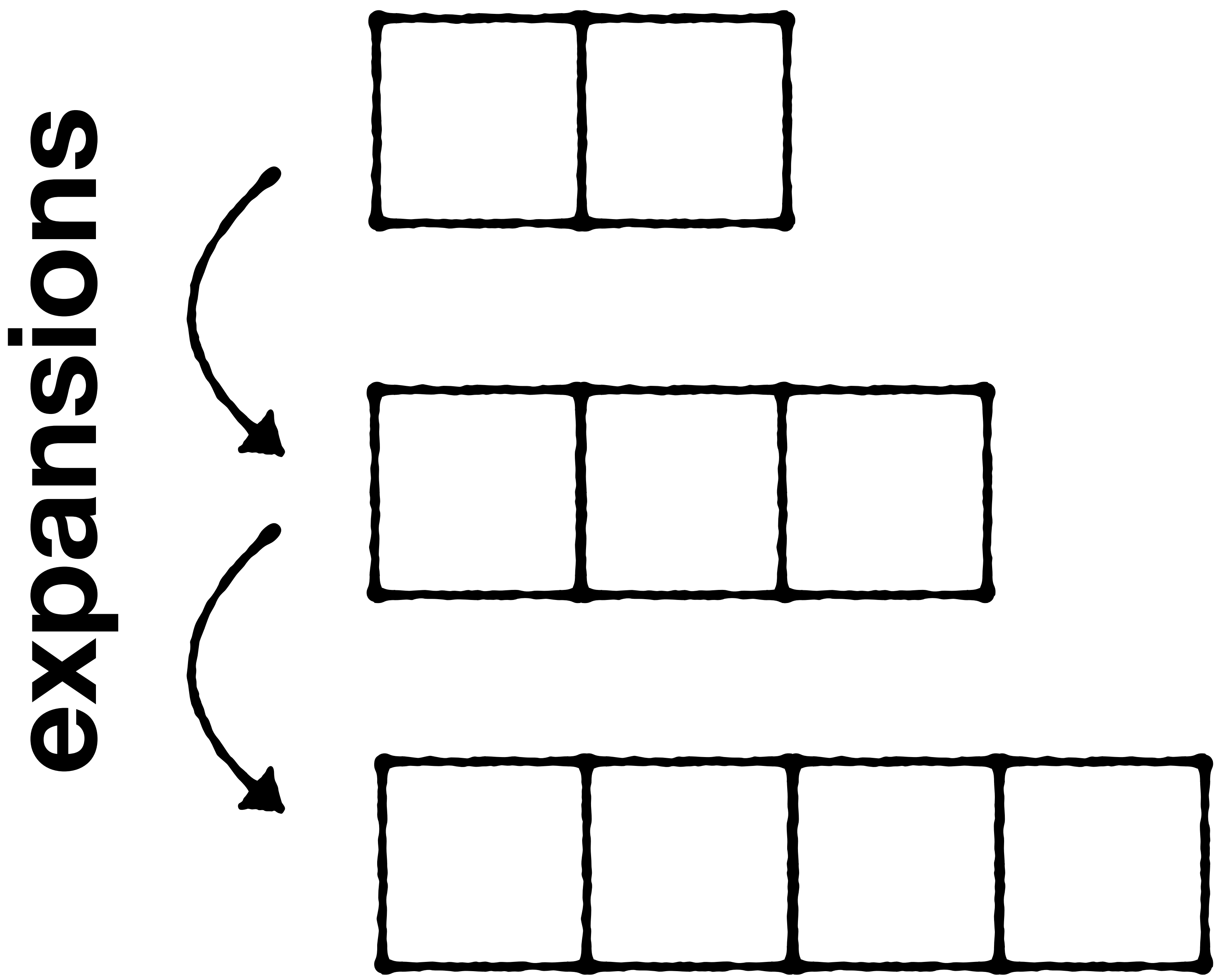


$$\text{Max Space-Amp} = \frac{\text{used + unused}}{\text{used}} = \frac{G^2}{G - 1} = 4 \quad \text{for } G = 2$$

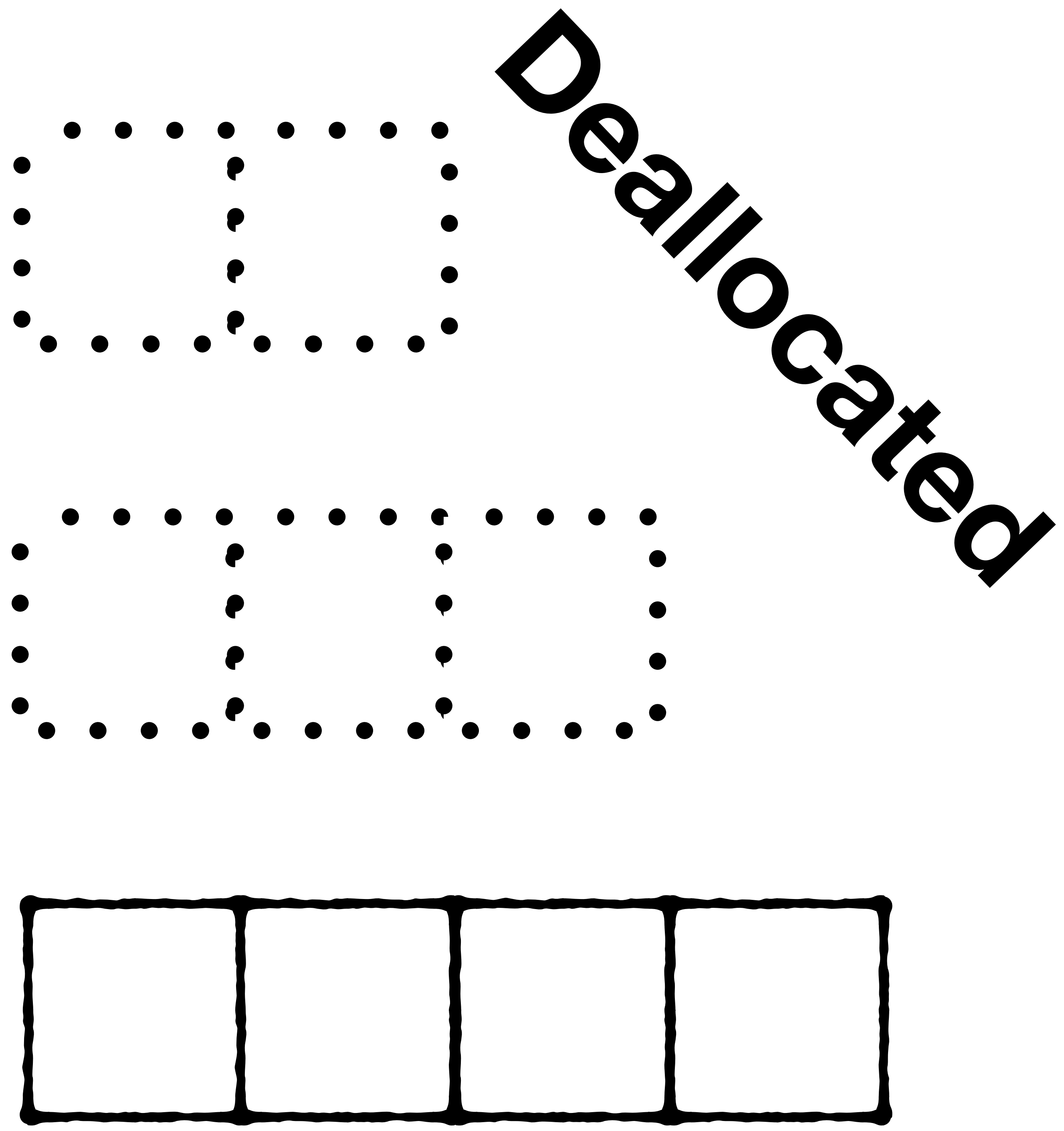


Suppose we use small size ratio, e.g., G = 1.2

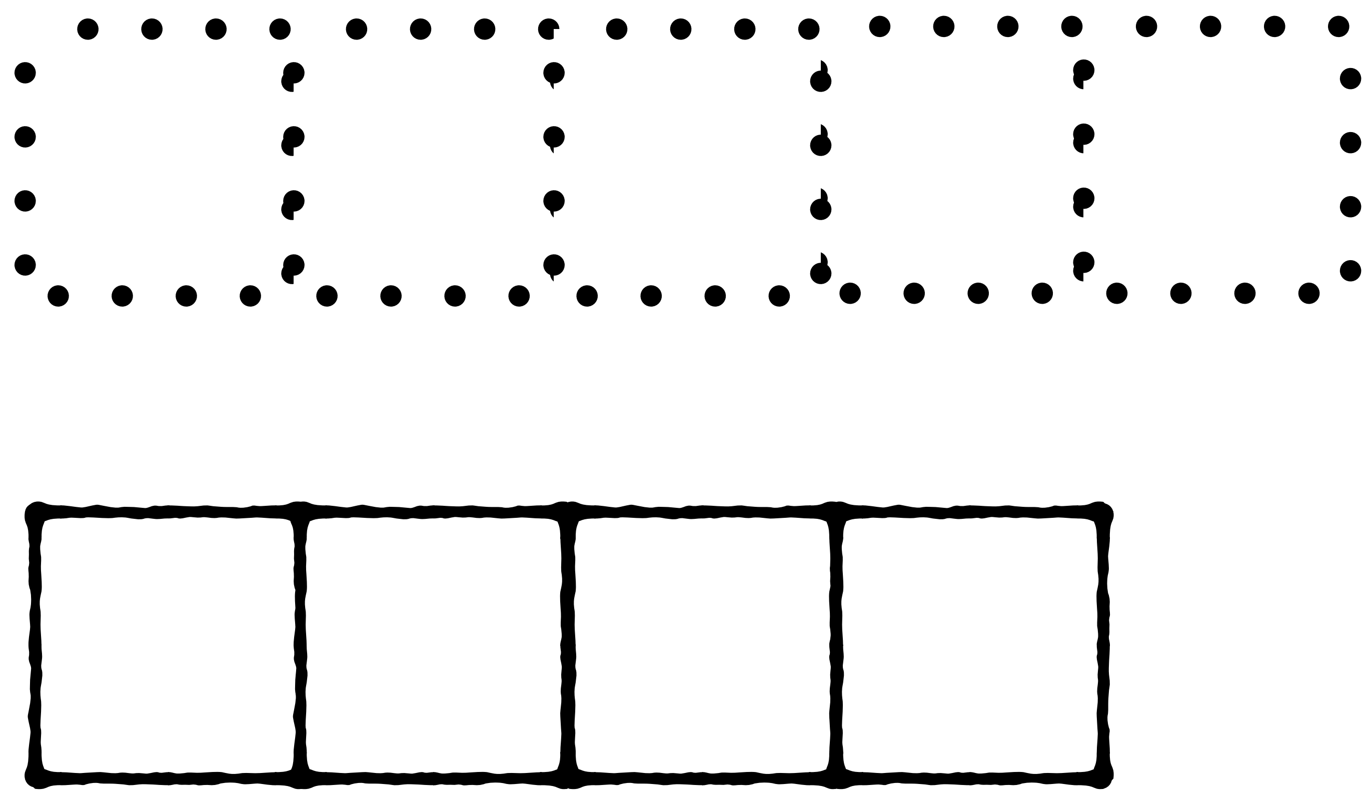
Suppose we use small size ratio,
e.g., G = 1.2



Suppose we use small size ratio, e.g., G = 1.2



Suppose we use small size ratio, e.g., G = 1.2

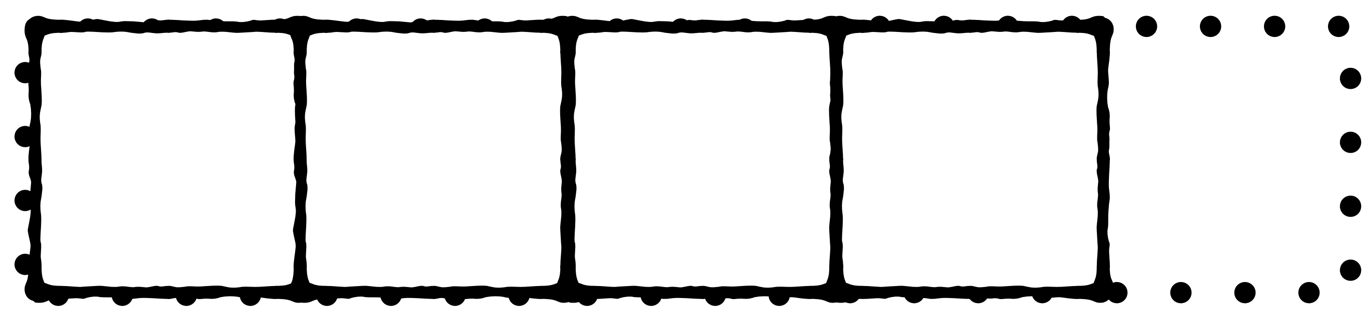


Total deallocated space

>

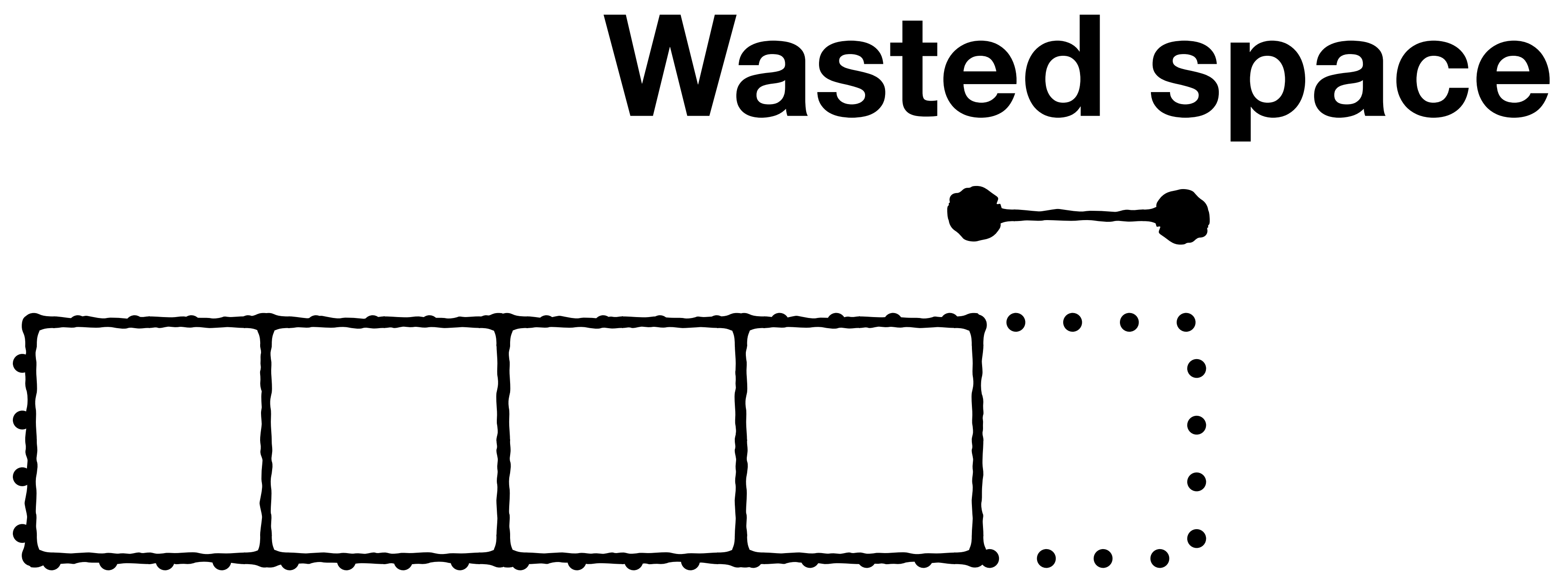
Total allocated space

Suppose we use small size ratio, e.g., G = 1.2

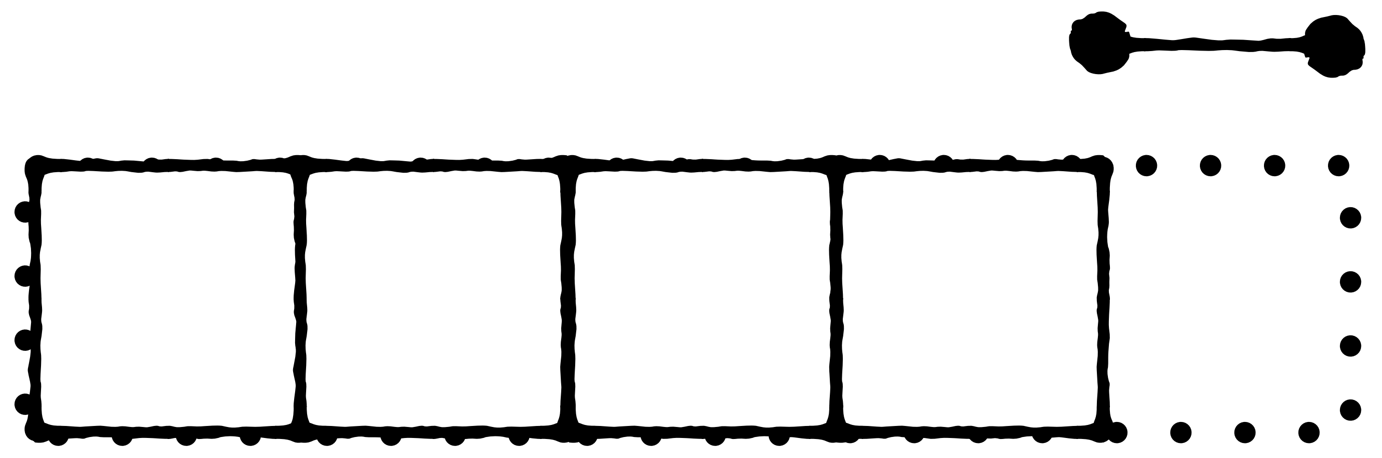


Reuse is possible

Suppose we use small size ratio, e.g., G = 1.2

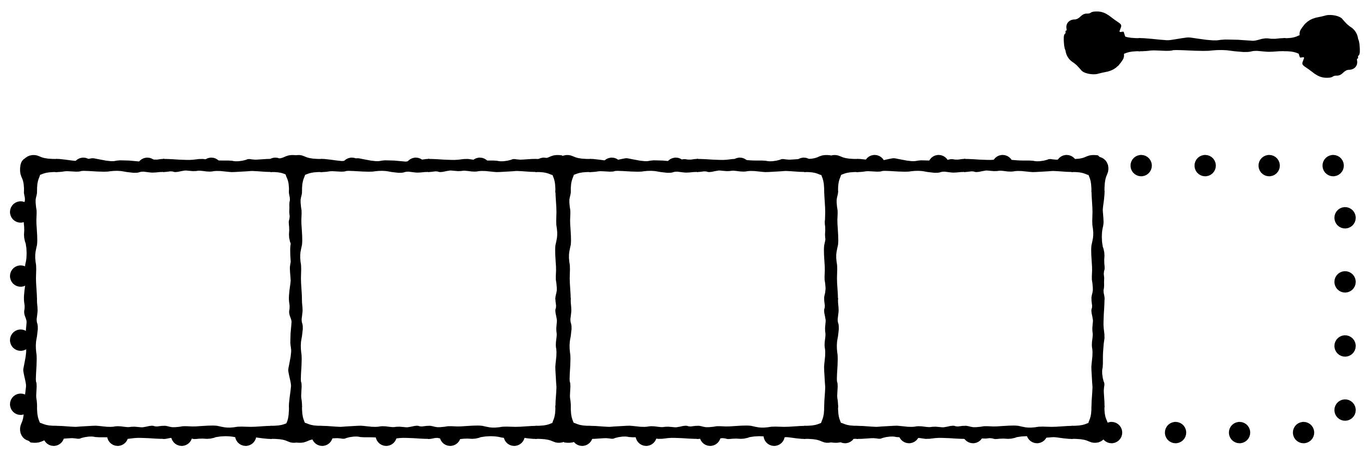


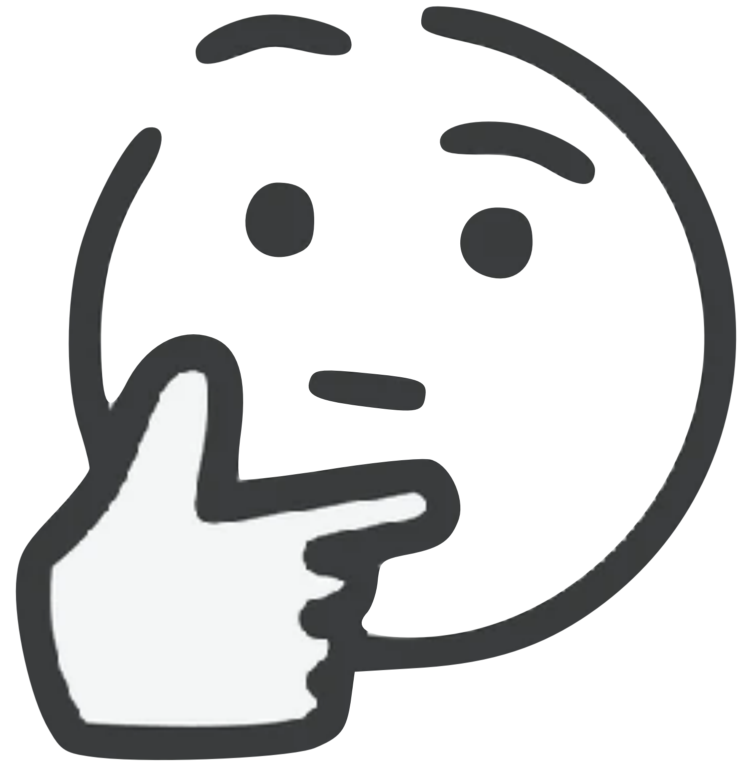
Suppose we use small size ratio, e.g., G = 1.2



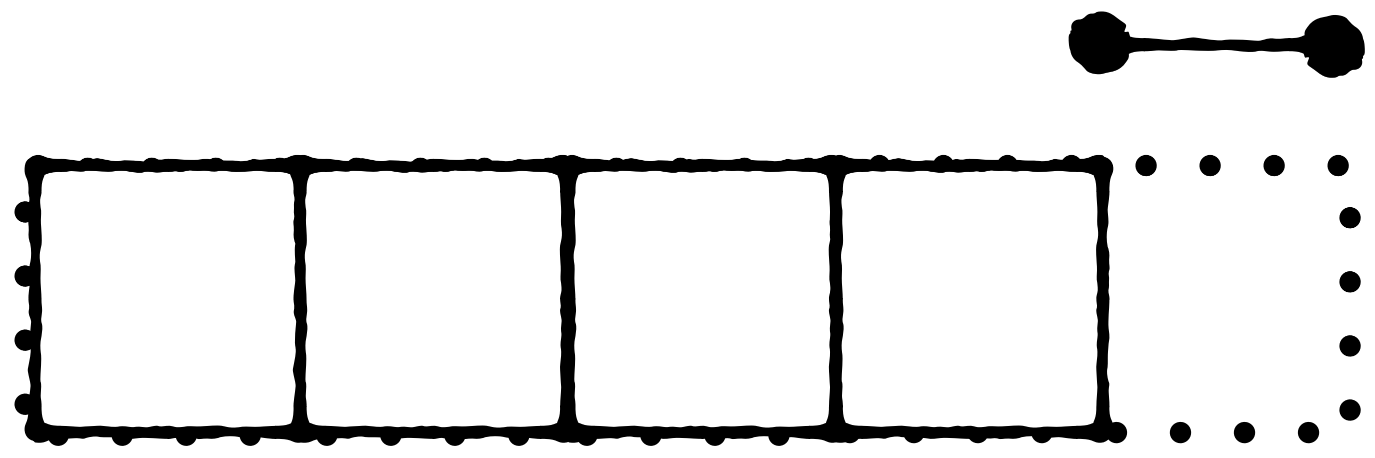
Could have expanded by larger factor

For which growth factor, do we perfectly reuse the space?





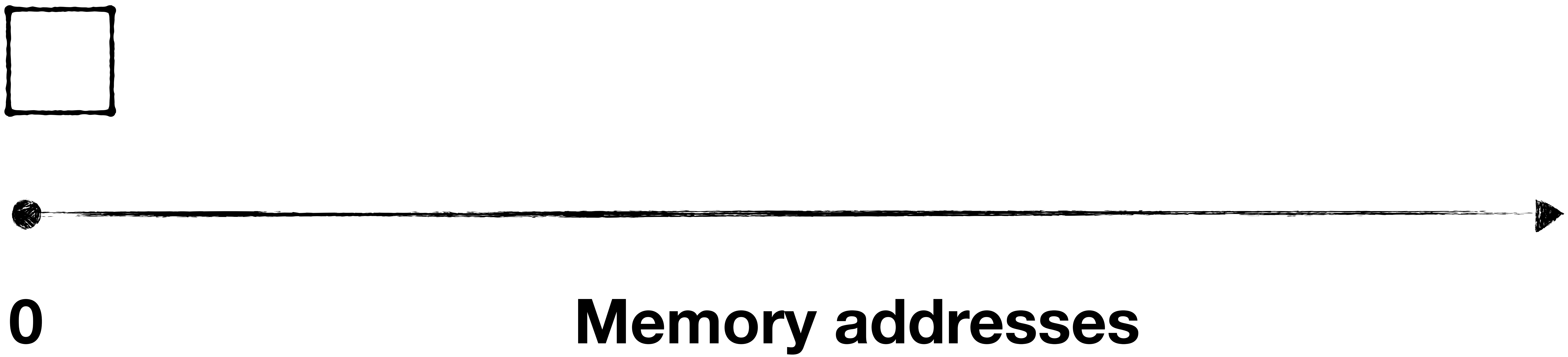
For which growth factor, do we perfectly reuse the space?



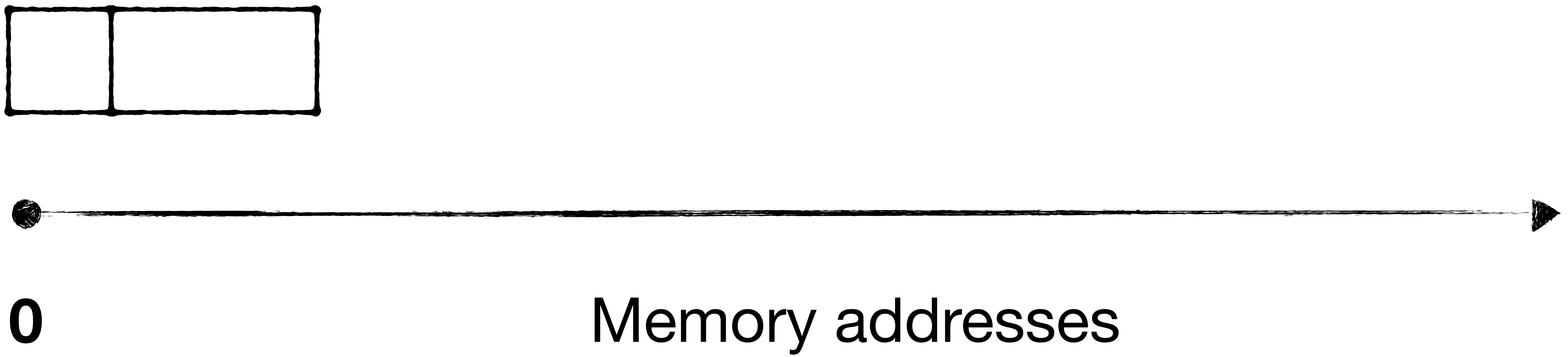
Applicable question across other data structures, e.g., hash tables

Assumptions on memory allocator

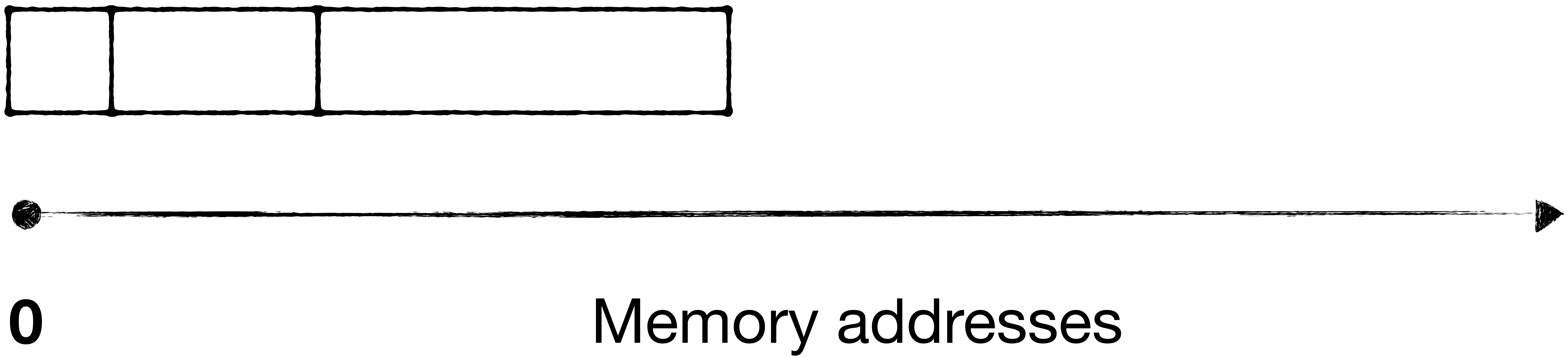
Assumptions: **Contiguous allocation**



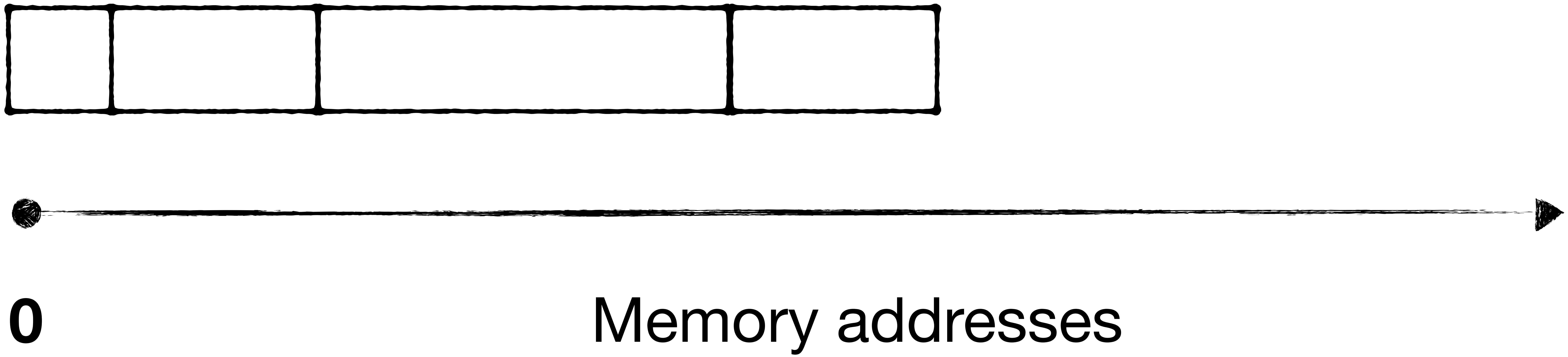
Assumptions: **Contiguous allocation**



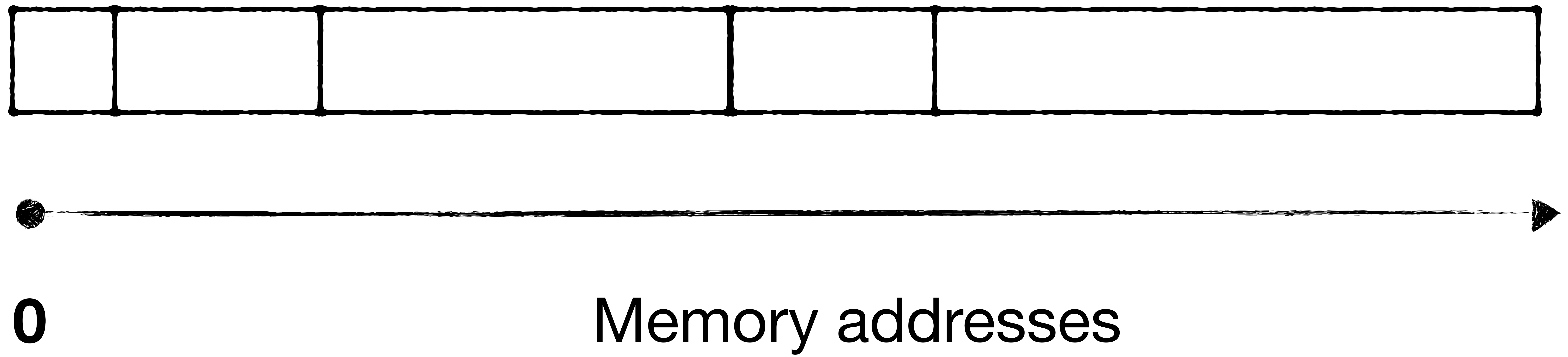
Assumptions: **Contiguous allocation**



Assumptions: **Contiguous allocation**

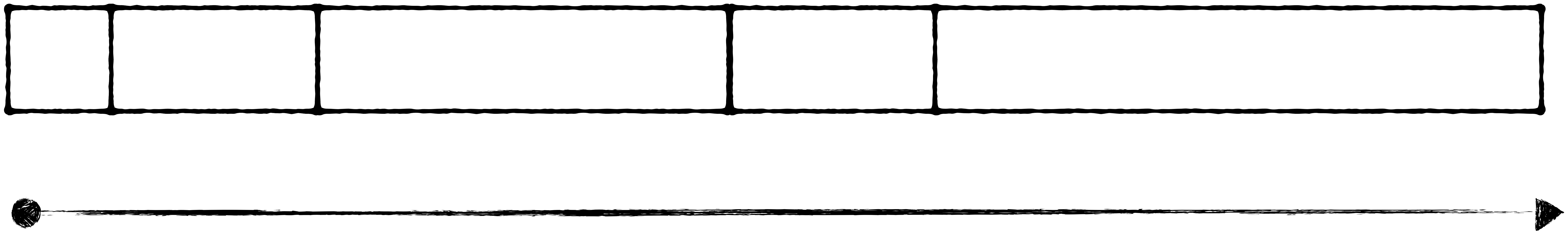


Assumptions: **Contiguous allocation**



Assumptions: Contiguous allocation

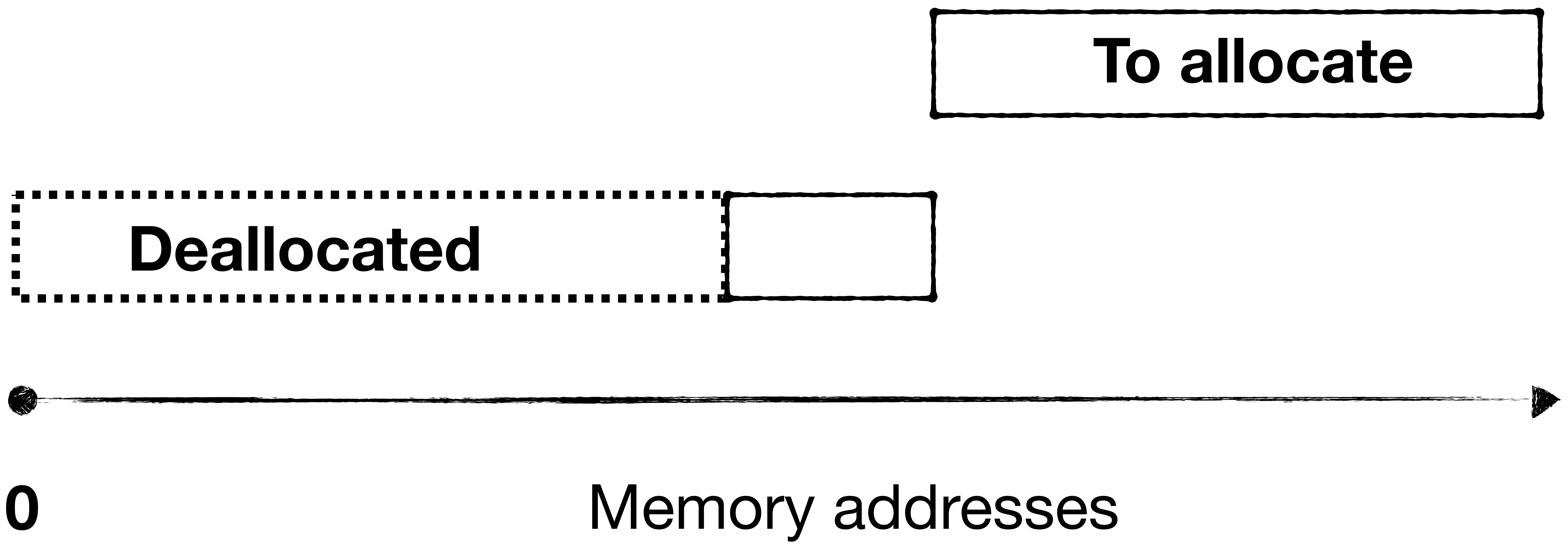
Reuse when possible



0 Memory addresses

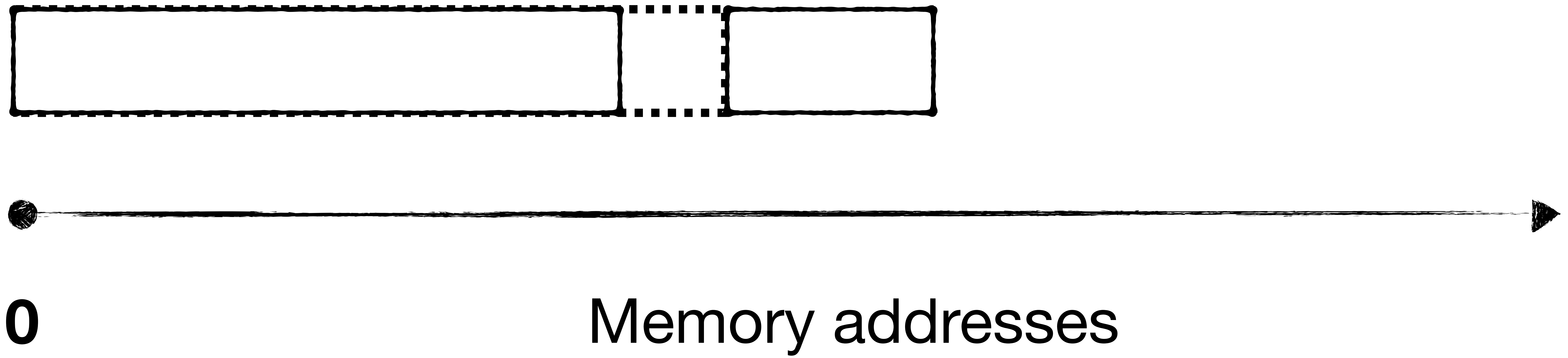
Assumptions: Contiguous allocation

Reuse when possible



Assumptions: Contiguous allocation

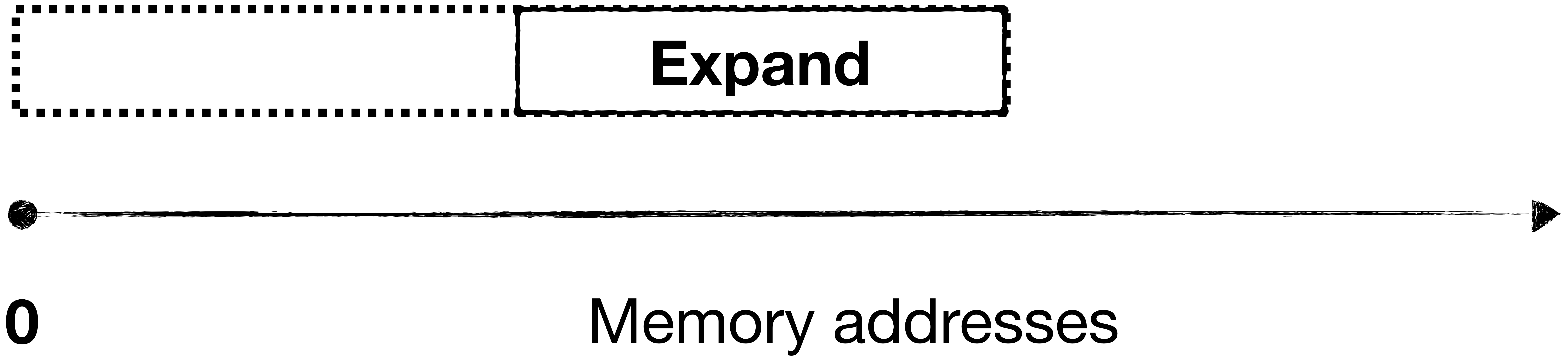
Reuse when possible



Assumptions: Contiguous allocation

Reuse when possible

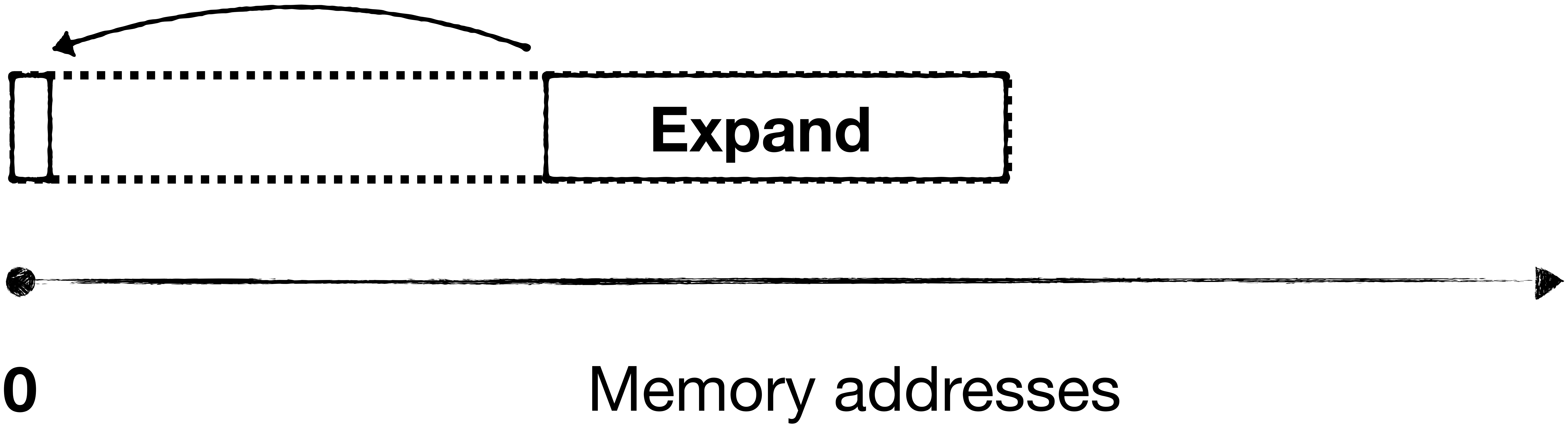
Deallocate as we copy (simplistic)



Assumptions: Contiguous allocation

Reuse when possible

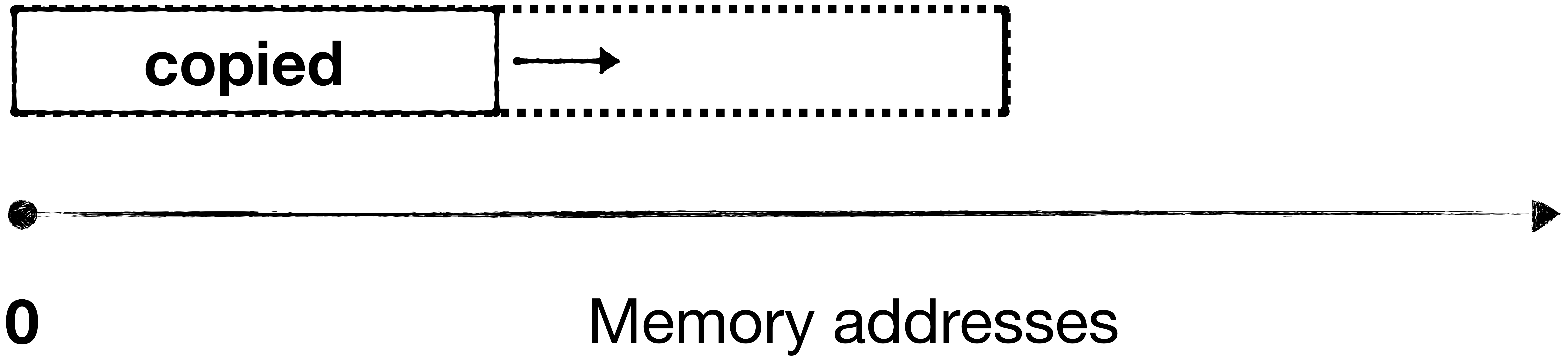
Deallocate as we copy (simplistic)



Assumptions: Contiguous allocation

Reuse when possible

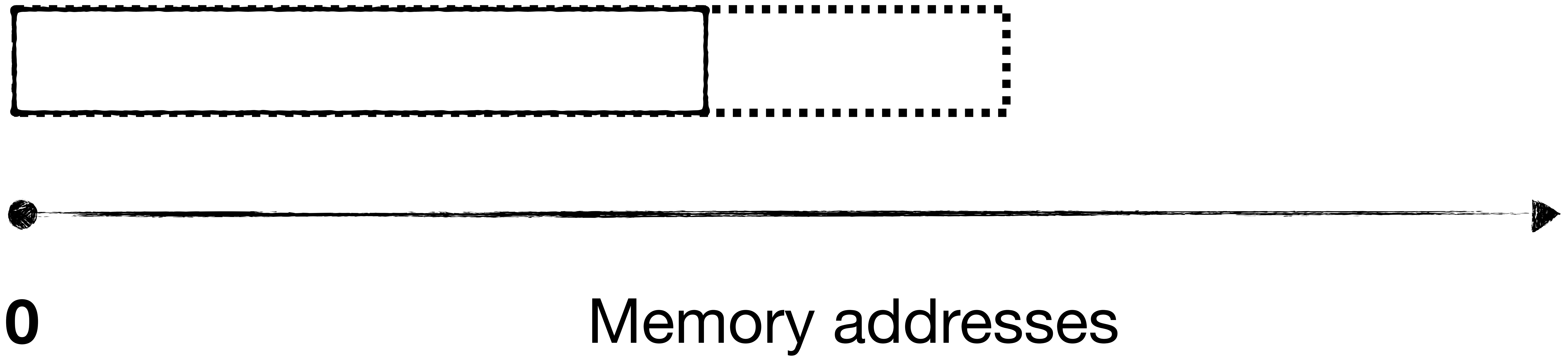
Deallocate as we copy (simplistic)



Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy (simplistic)

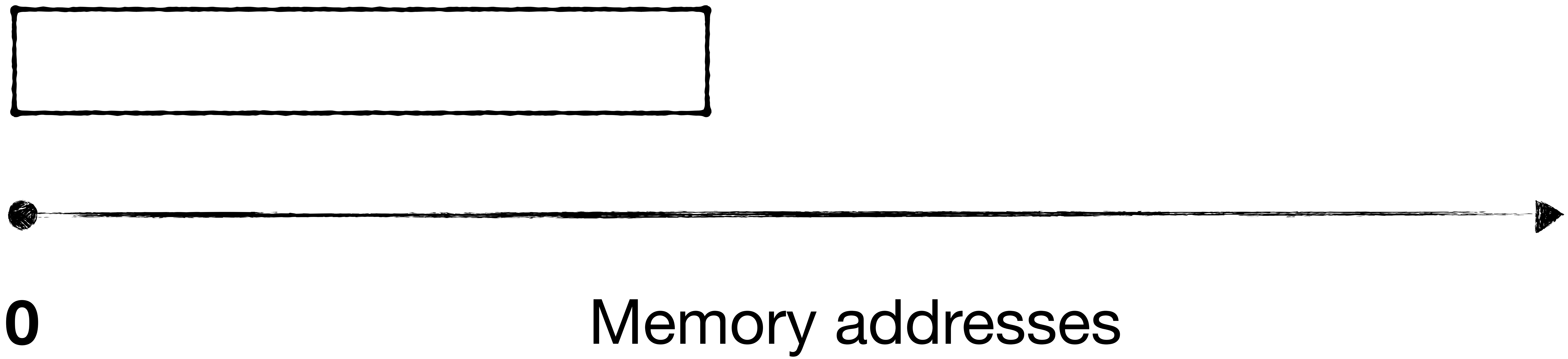


Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can’t expand in-place

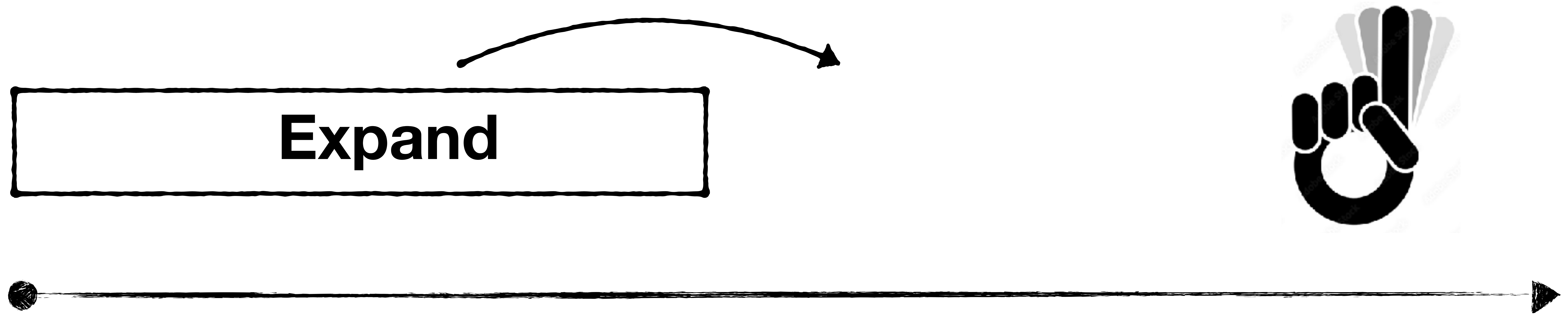


Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can’t expand in-place



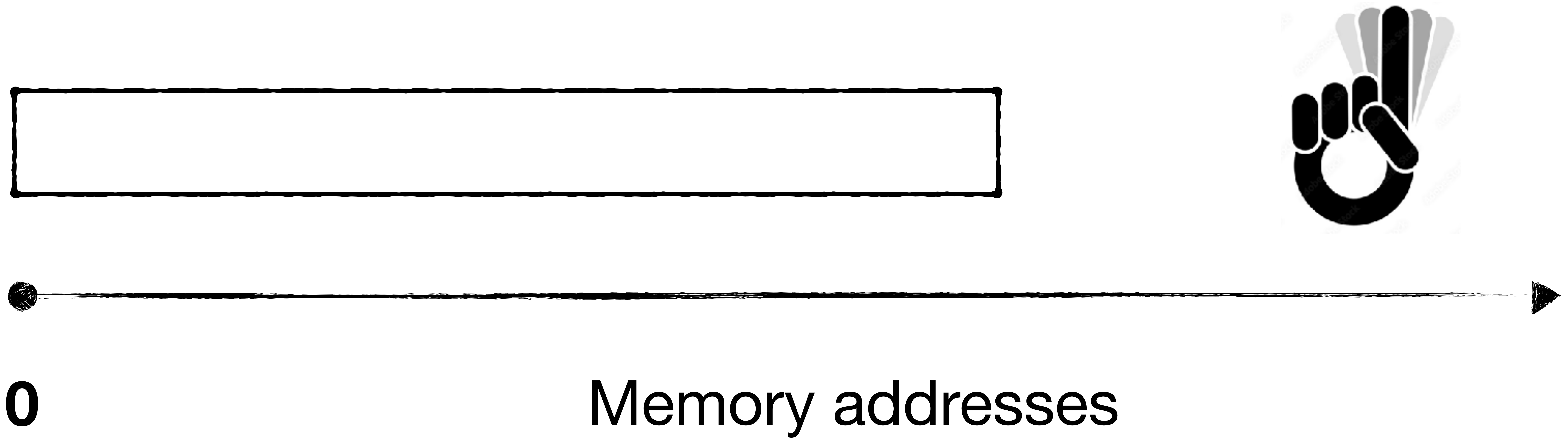
0 Memory addresses

Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can’t expand in-place

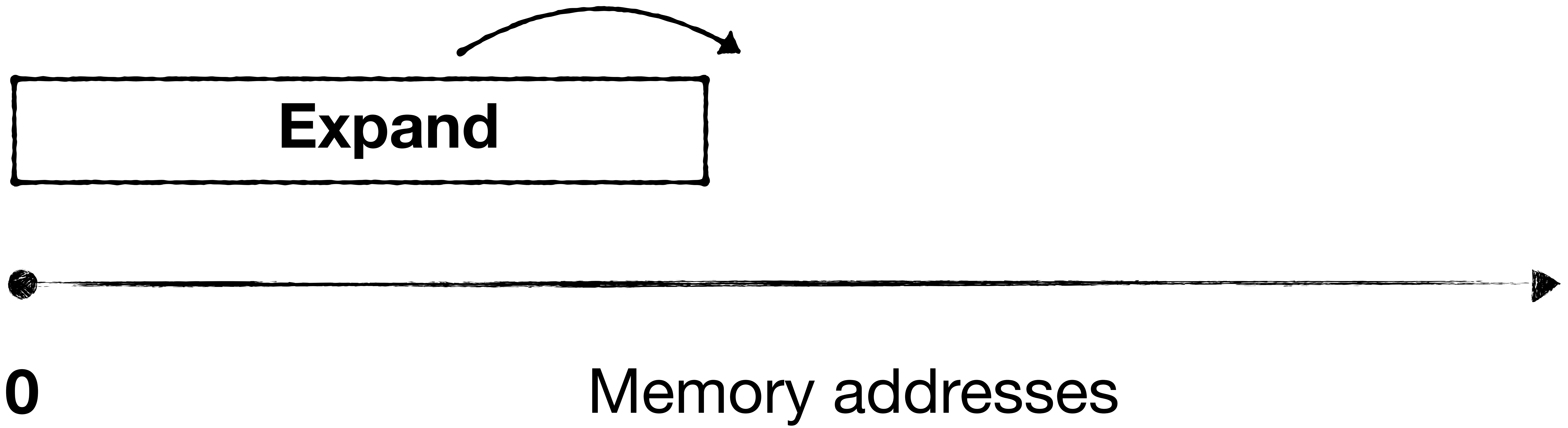


Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can't expand in-place

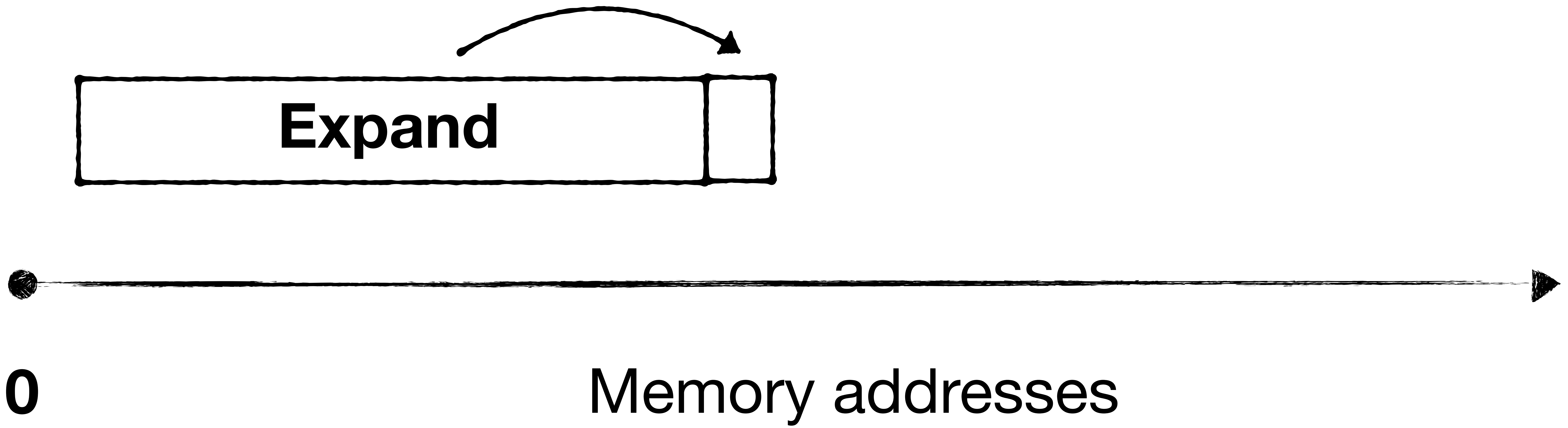


Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can’t expand in-place

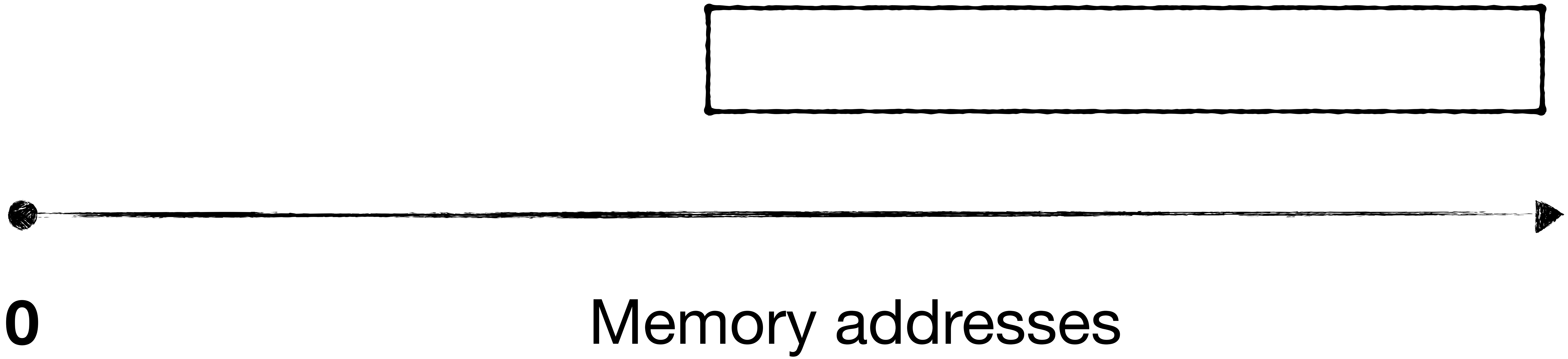


Assumptions: Contiguous allocation

Reuse when possible

Deallocate as we copy

Can’t expand in-place



For which growth factor, do we perfectly reuse the space?

For which growth factor, do we perfectly reuse the space?

$$\textbf{Size}_{i-2} + \textbf{Size}_{i-1} = \textbf{Size}_{i}$$

For which growth factor, do we perfectly reuse the space?

$$\mathbf{Size}_{i-2} + \mathbf{Size}_{i-1} = \mathbf{Size}_{i}$$

Subject to:

$$\frac{\mathbf{Size}_{i-1}}{\mathbf{Size}_{i-2}} = \frac{\mathbf{Size}_{i}}{\mathbf{Size}_{i-1}} = \mathbf{G}$$

For which growth factor, do we perfectly reuse the space?

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_i$$

Subject to:

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_i}{\text{Size}_{i-1}} = G$$

Clever ideas?

Fibonacci Series

1 1 2 3 5 8 13 21

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1170 - 1250

Italy

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 + 1 = 2 3 5 8 13 21

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1   1 + 2 = 3   5   8   13   21

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_i$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_i}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 + 3 = 5 8 13 21

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

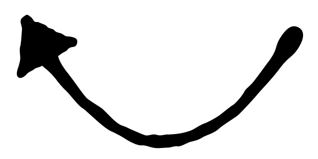
1 1 2 + 3 = 5 8 13 21

Satisfies this: → $\mathbf{Size}_{i-2} + \mathbf{Size}_{i-1} = \mathbf{Size}_{i}$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21



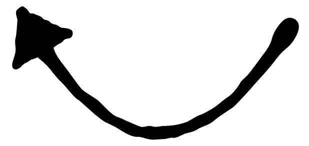
1

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21



2

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.5

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.666

$$Size_{i-2} + Size_{i-1} = Size_{i}$$

$$\frac{Size_{i-1}}{Size_{i-2}} = \frac{Size_{i}}{Size_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.6

$$Size_{i-2} + Size_{i-1} = Size_{i}$$

$$\frac{Size_{i-1}}{Size_{i-2}} = \frac{Size_{i}}{Size_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.625

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.625

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21

1.615

$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21 34 55 **89** **144**

1.618

$$Size_{i-2} + Size_{i-1} = Size_{i}$$

$$\frac{Size_{i-1}}{Size_{i-2}} = \frac{Size_{i}}{Size_{i-1}} = G$$

Fibonacci Series

1 1 2 3 5 8 13 21 34 55 89 144

1.618

Ratio converges to the “Golden Ratio” ϕ

Fibonacci Series

1 1 2 3 5 8 13 21 34 55 89 144

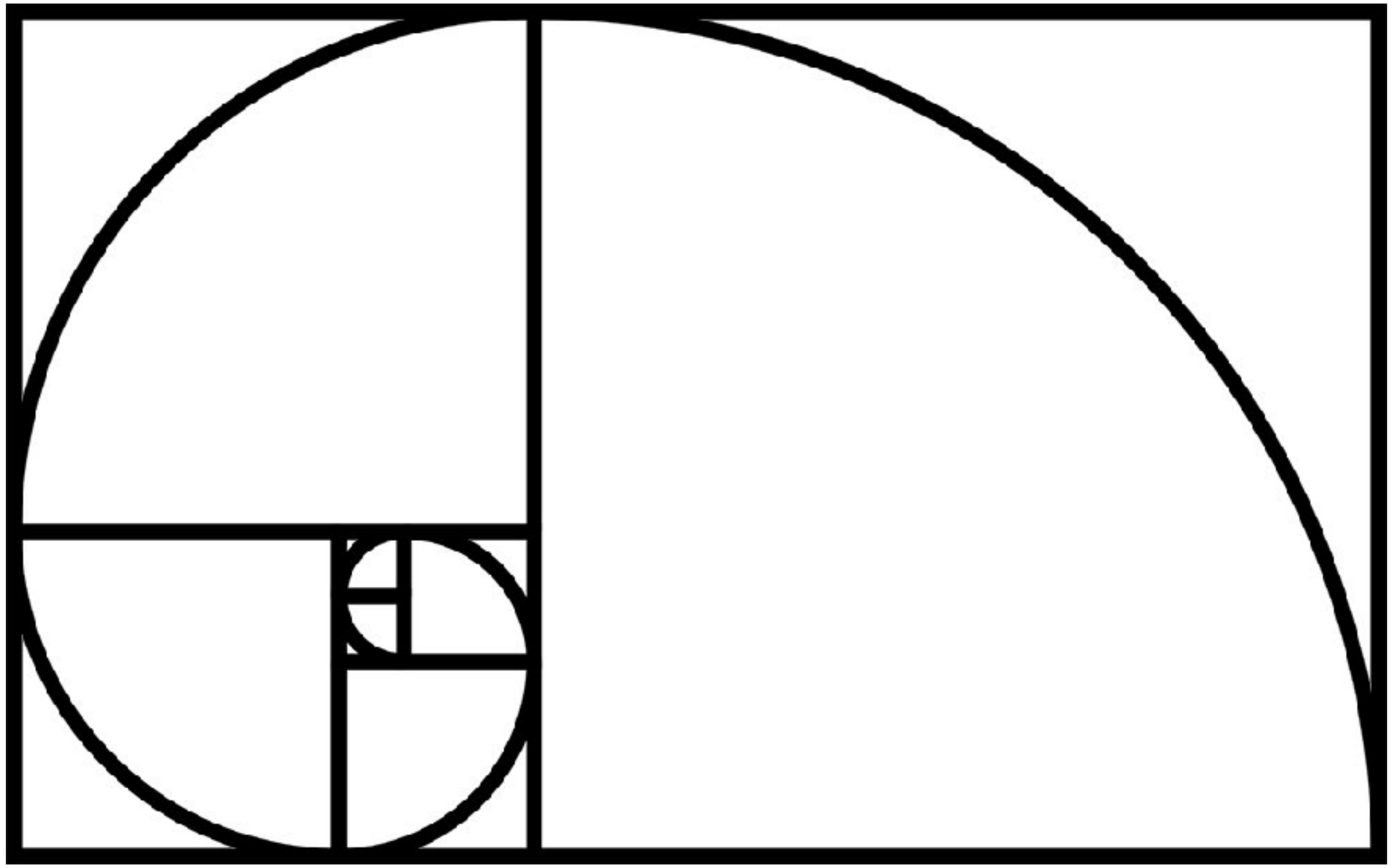
Ratio converges to the “Golden Ratio” ϕ = **1.618**033988749....

Fibonacci Series

1 1 2 3 5 8 13 21 34 55 89 144

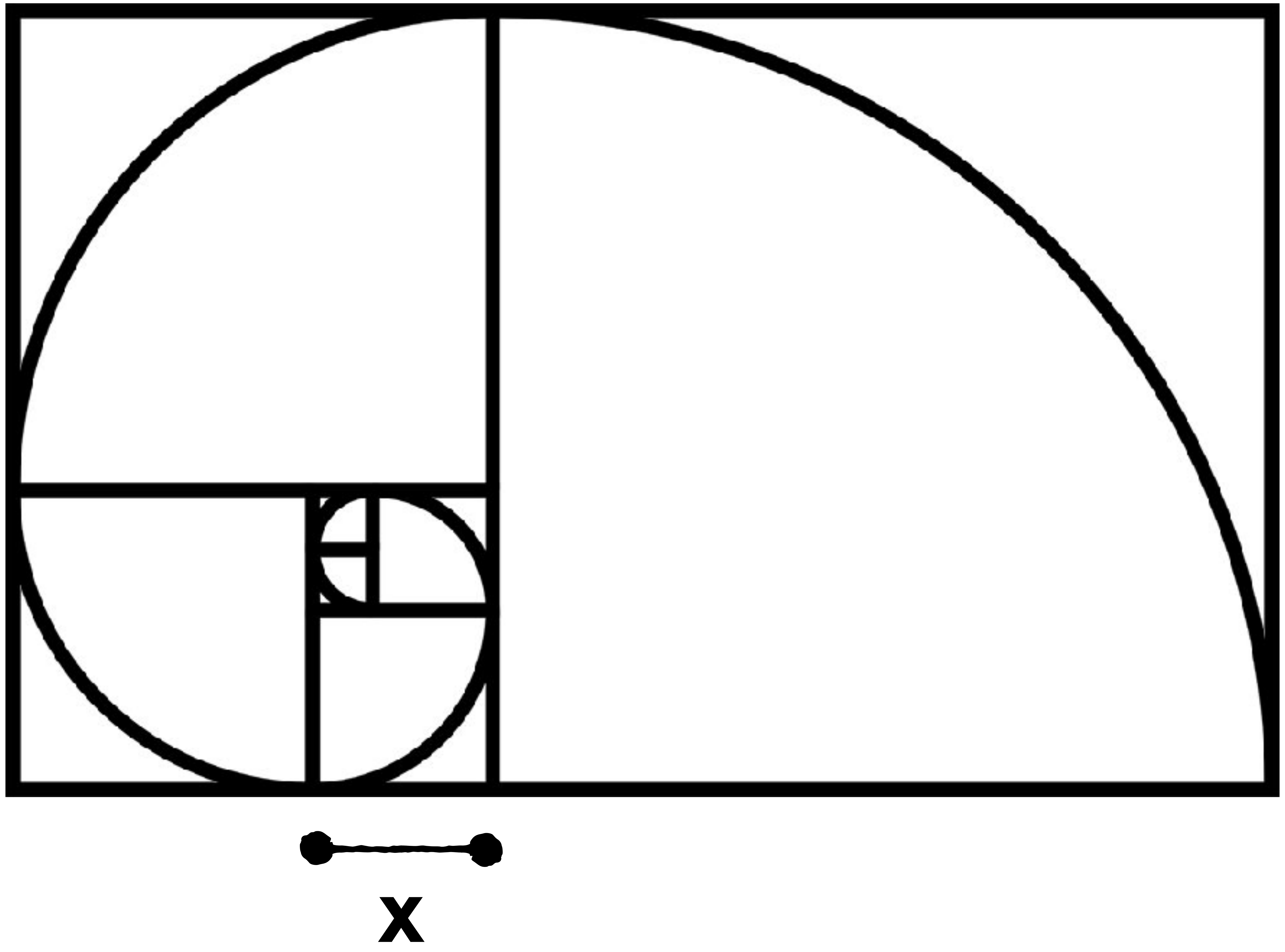
Ratio converges to the "Golden Ratio" $\phi = \dfrac{1+\sqrt{5}}{2}$

Golden Spiral



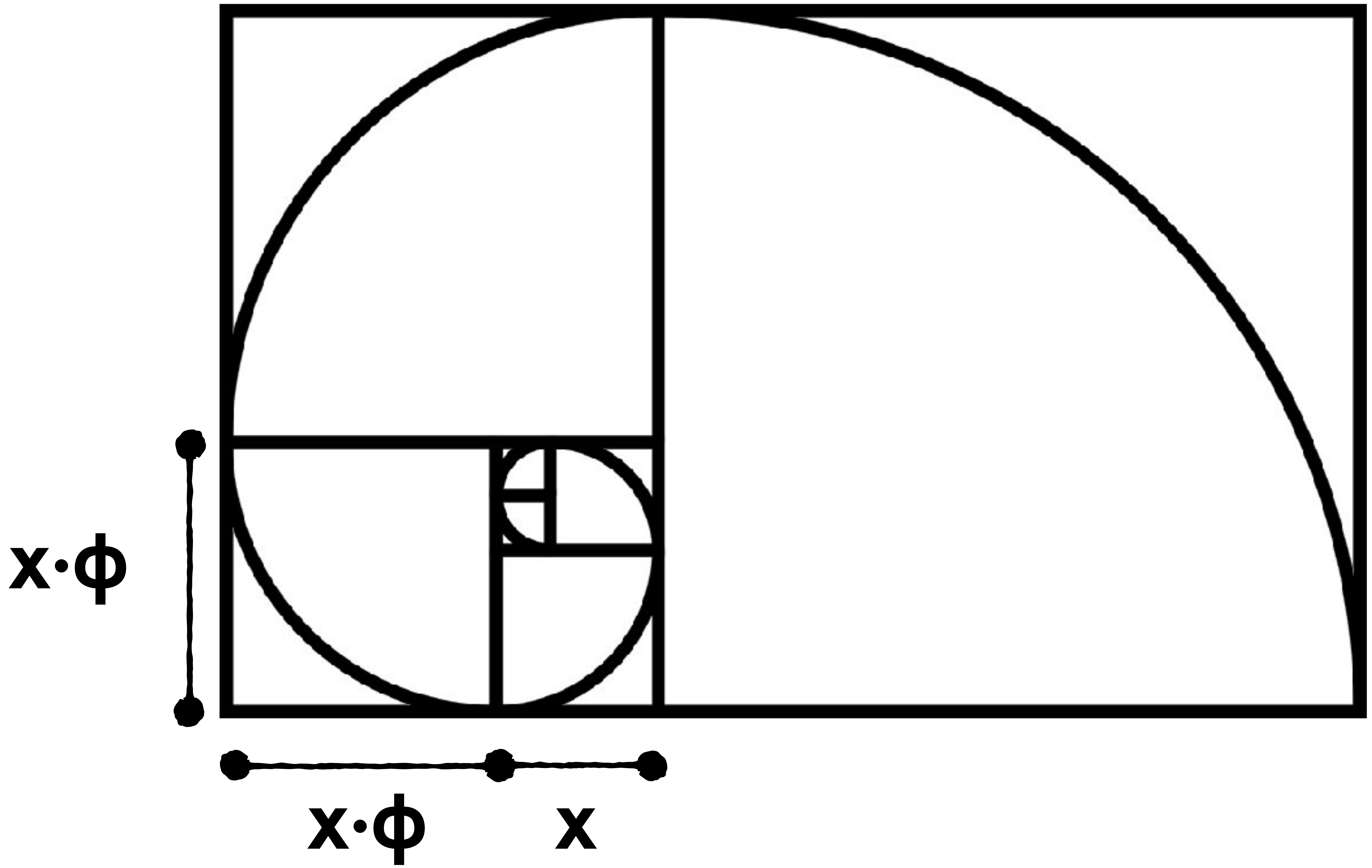
Ratio converges to the "Golden Ratio" $\phi = \frac{1+\sqrt{5}}{2}$

Golden Spiral

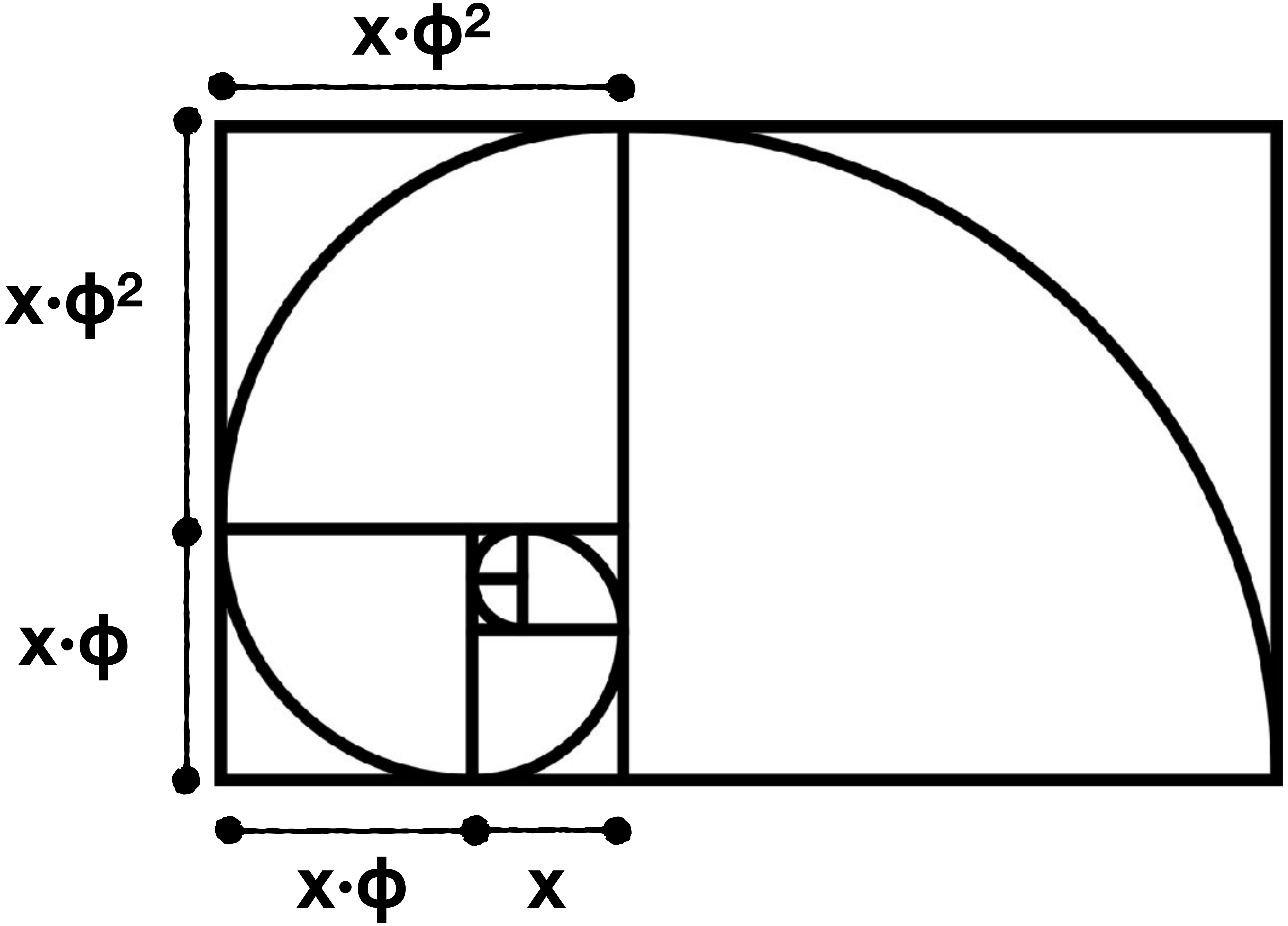


Ratio converges to the "Golden Ratio" $\phi = \frac{1+\sqrt{5}}{2}$

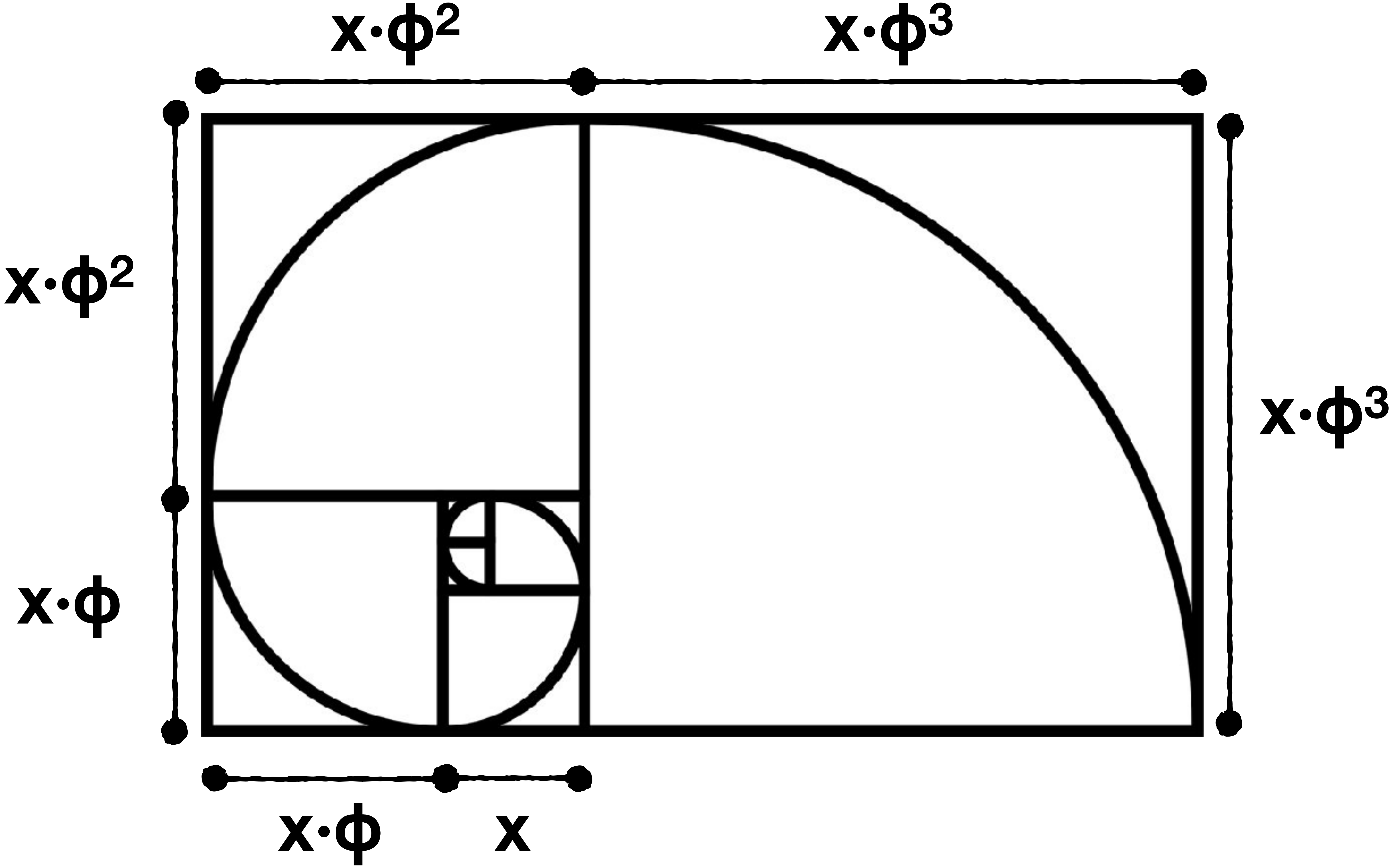
Golden Spiral



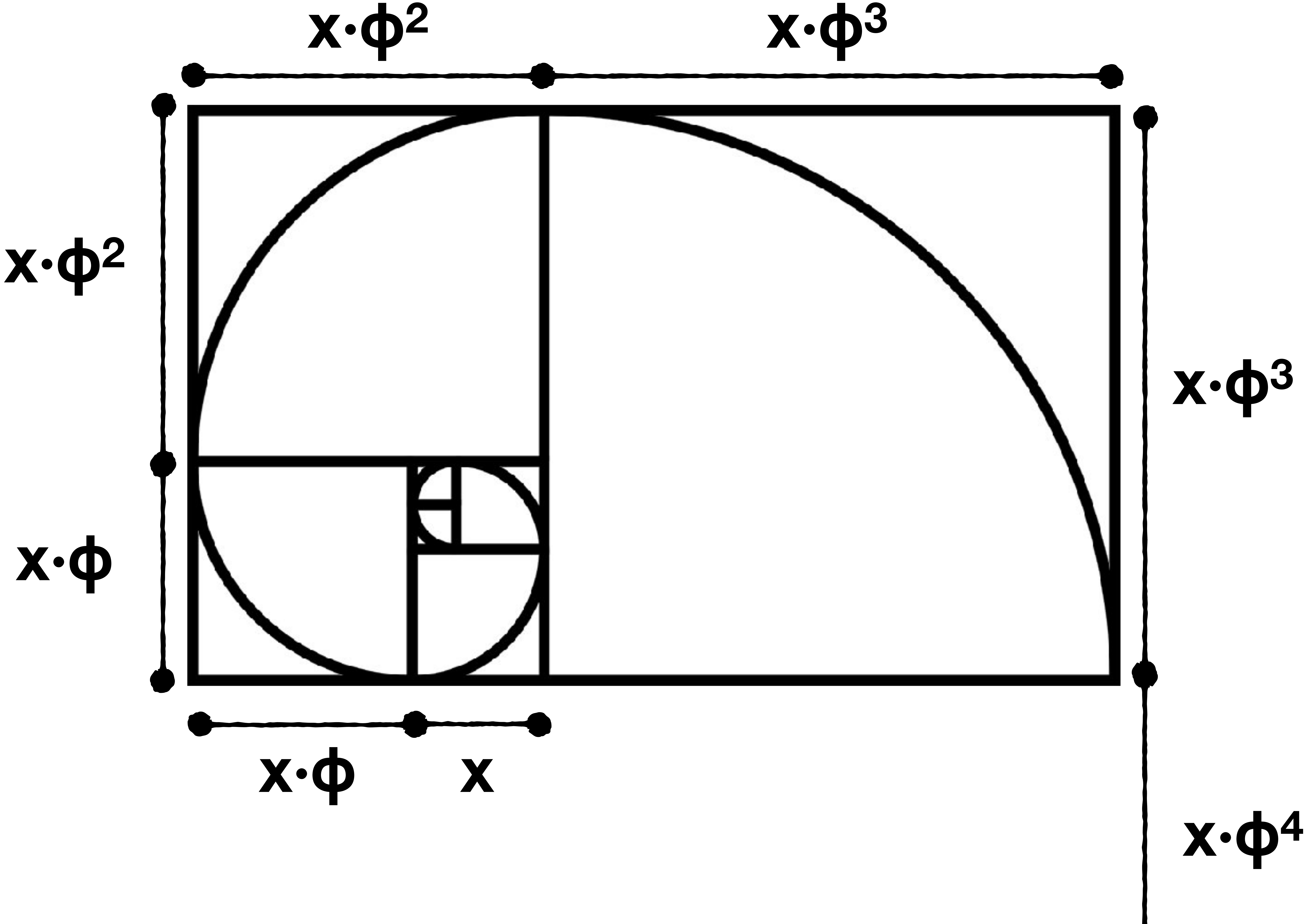
Golden Spiral



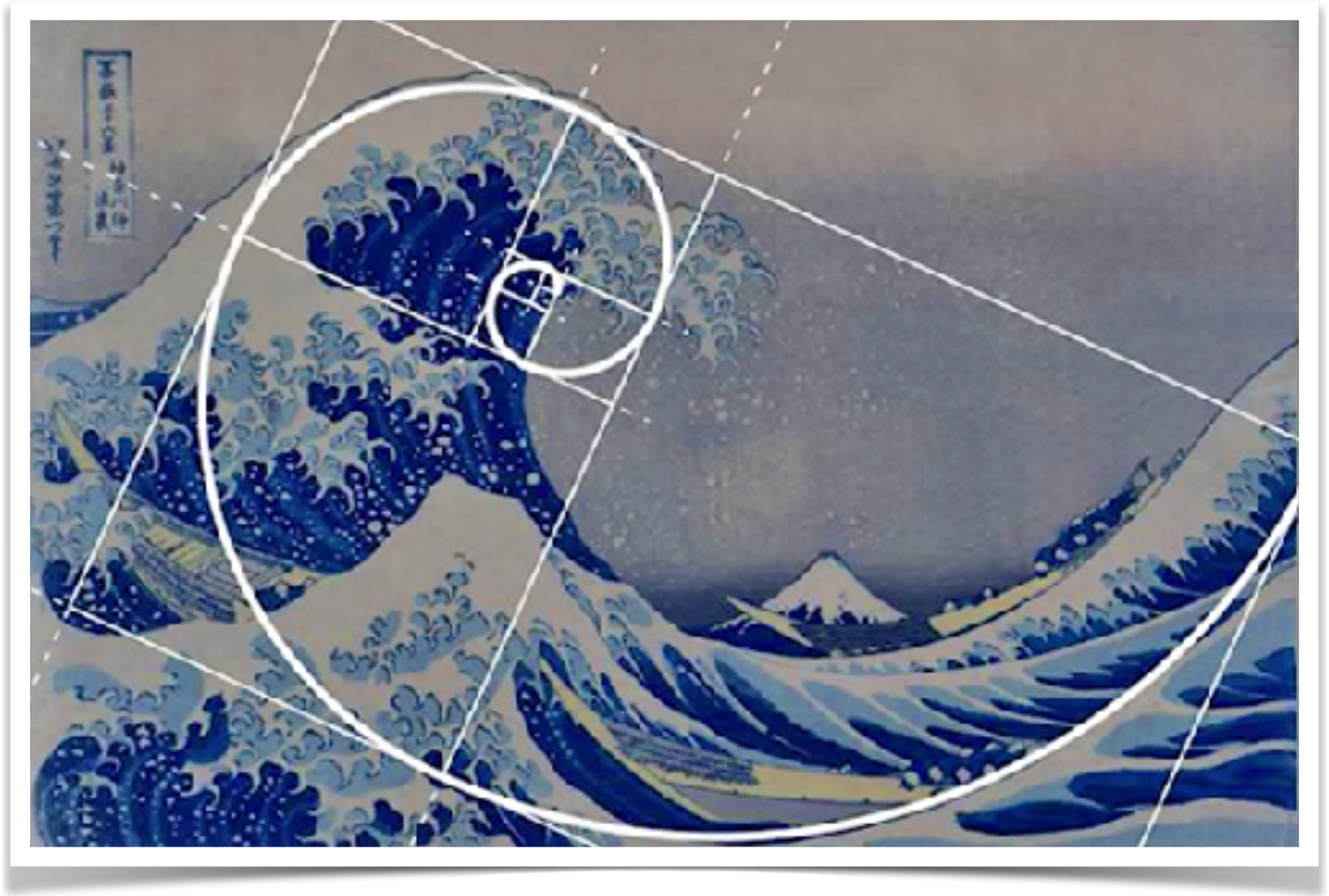
Golden Spiral



Golden Spiral



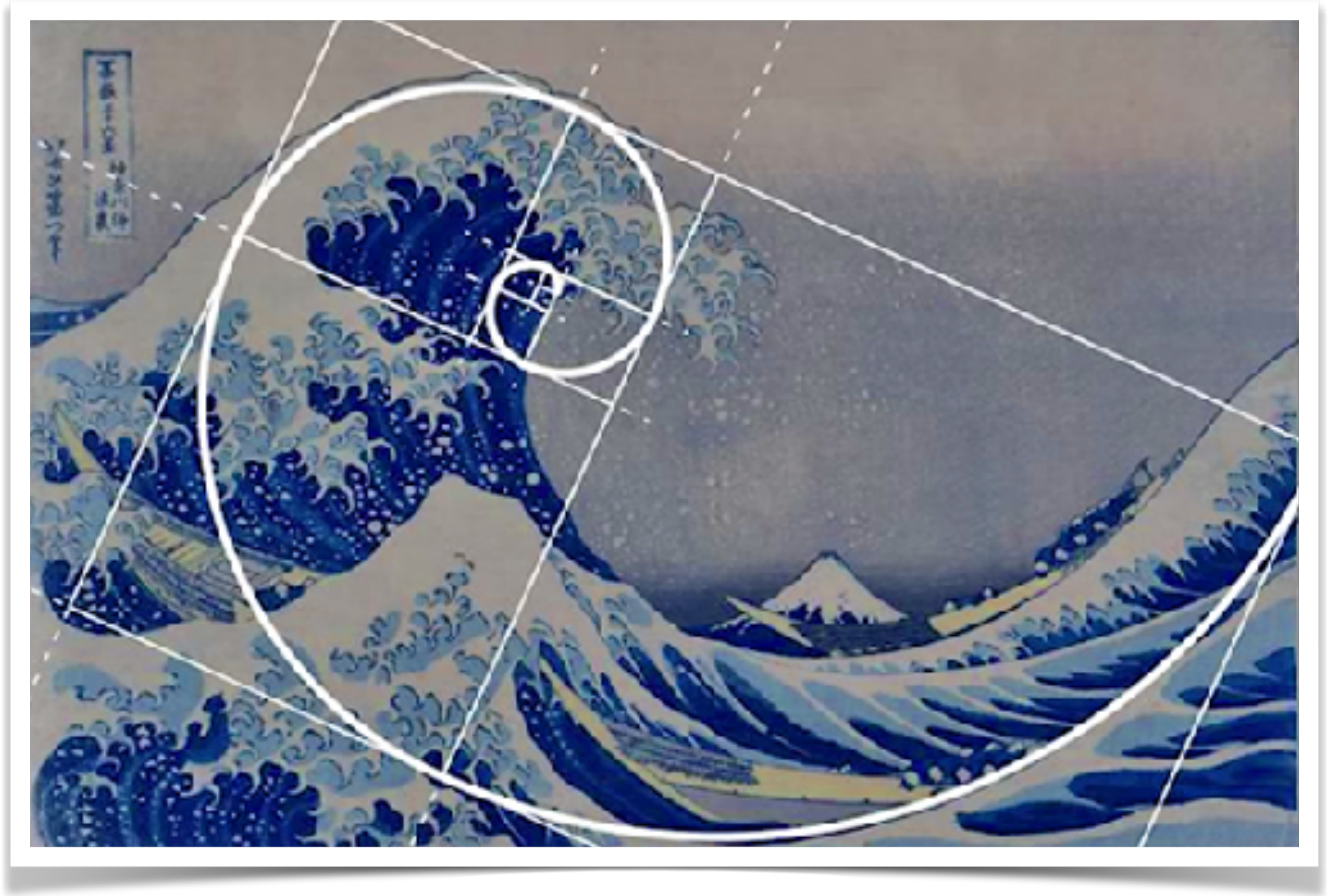
Golden Spiral

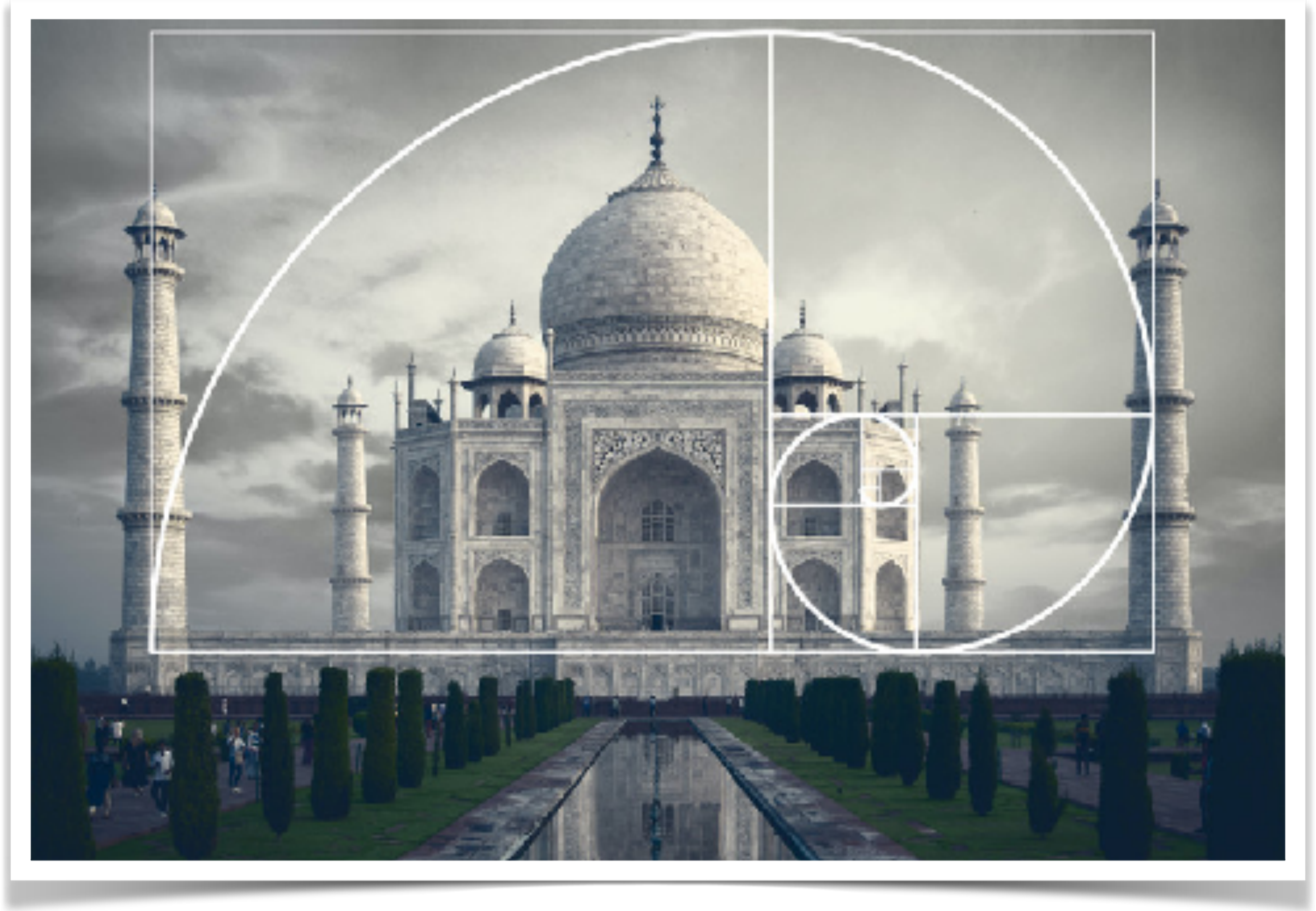


Art

The Great Wave off Kanagawa

Golden Spiral





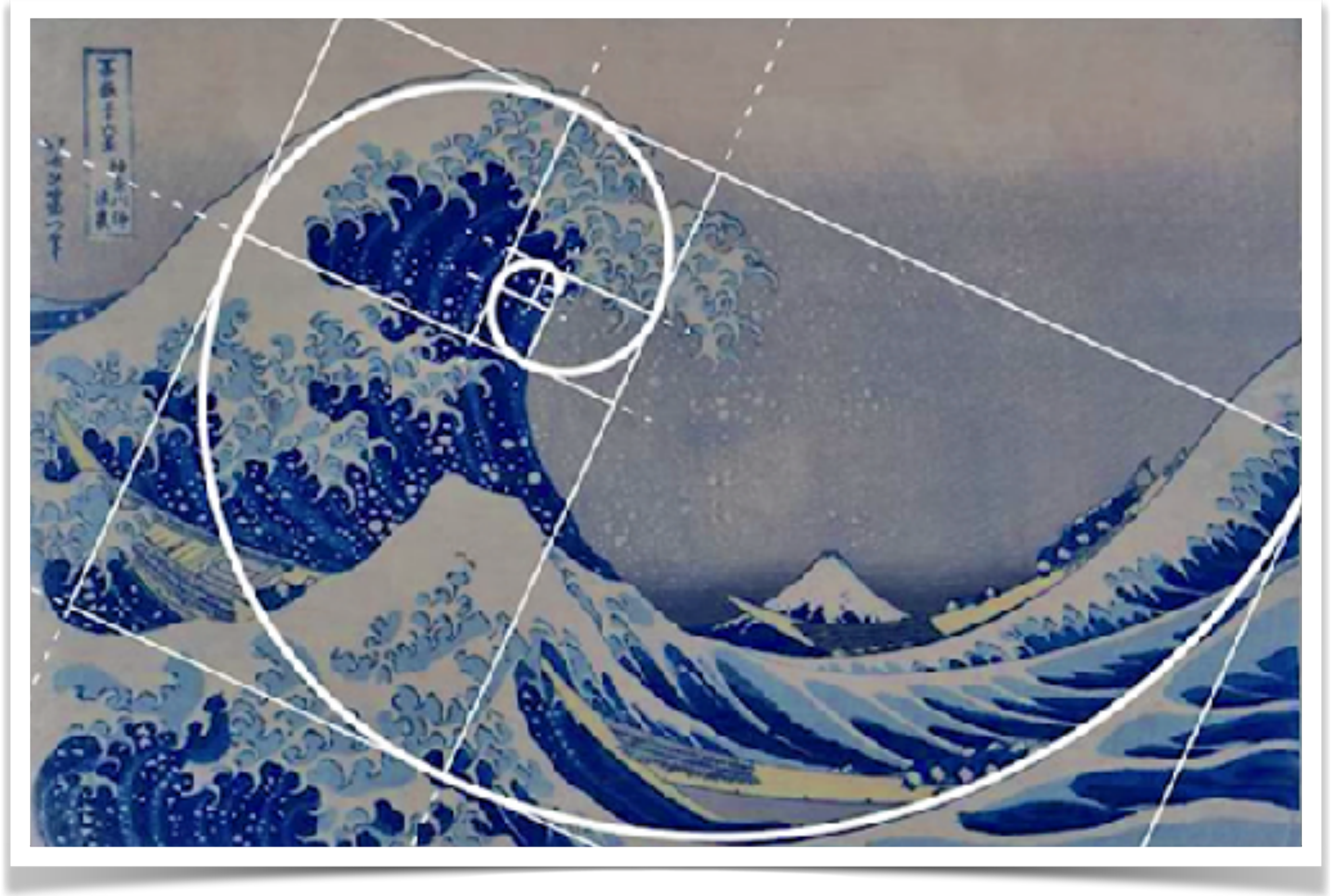
Art

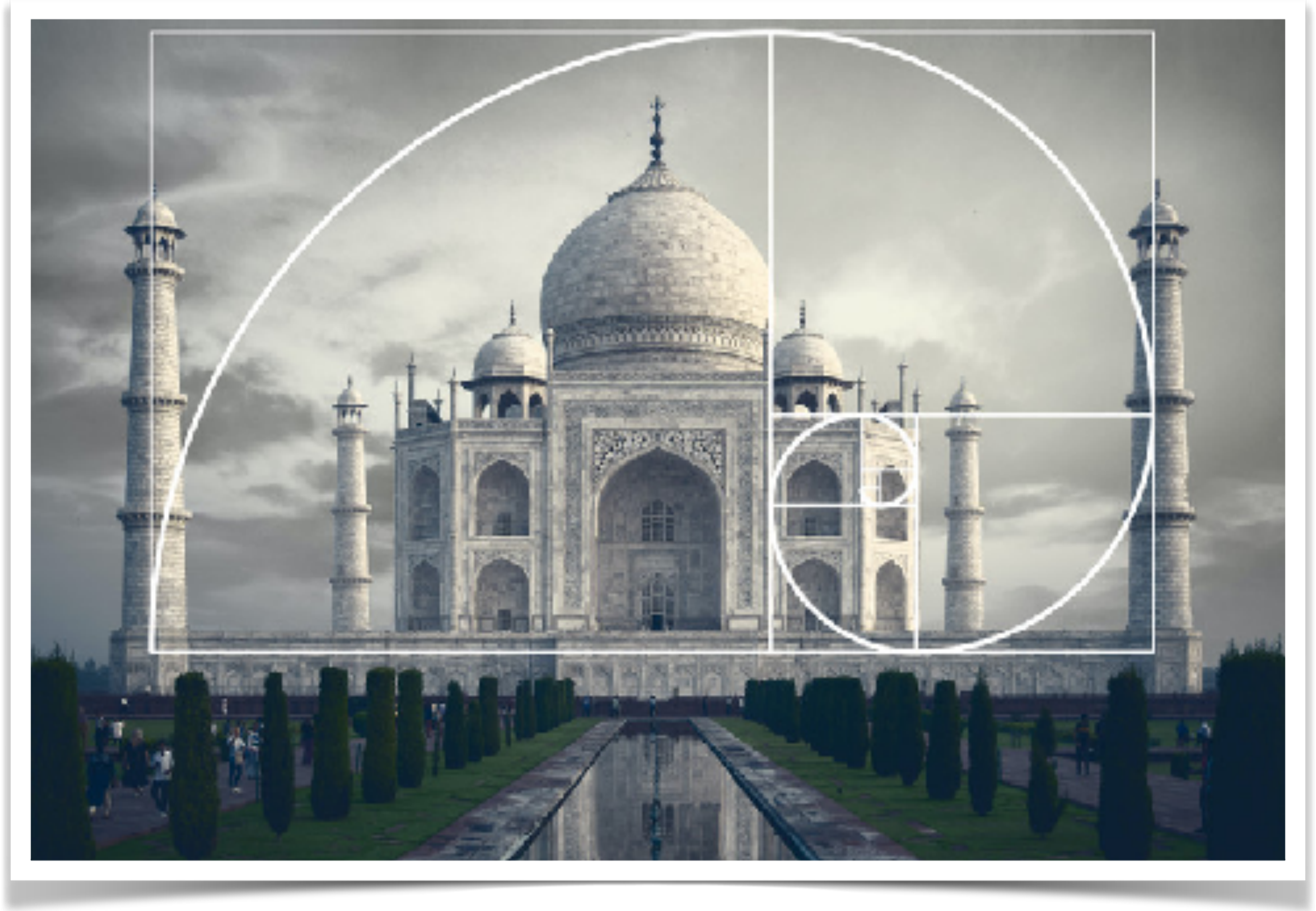
The Great Wave off Kanagawa

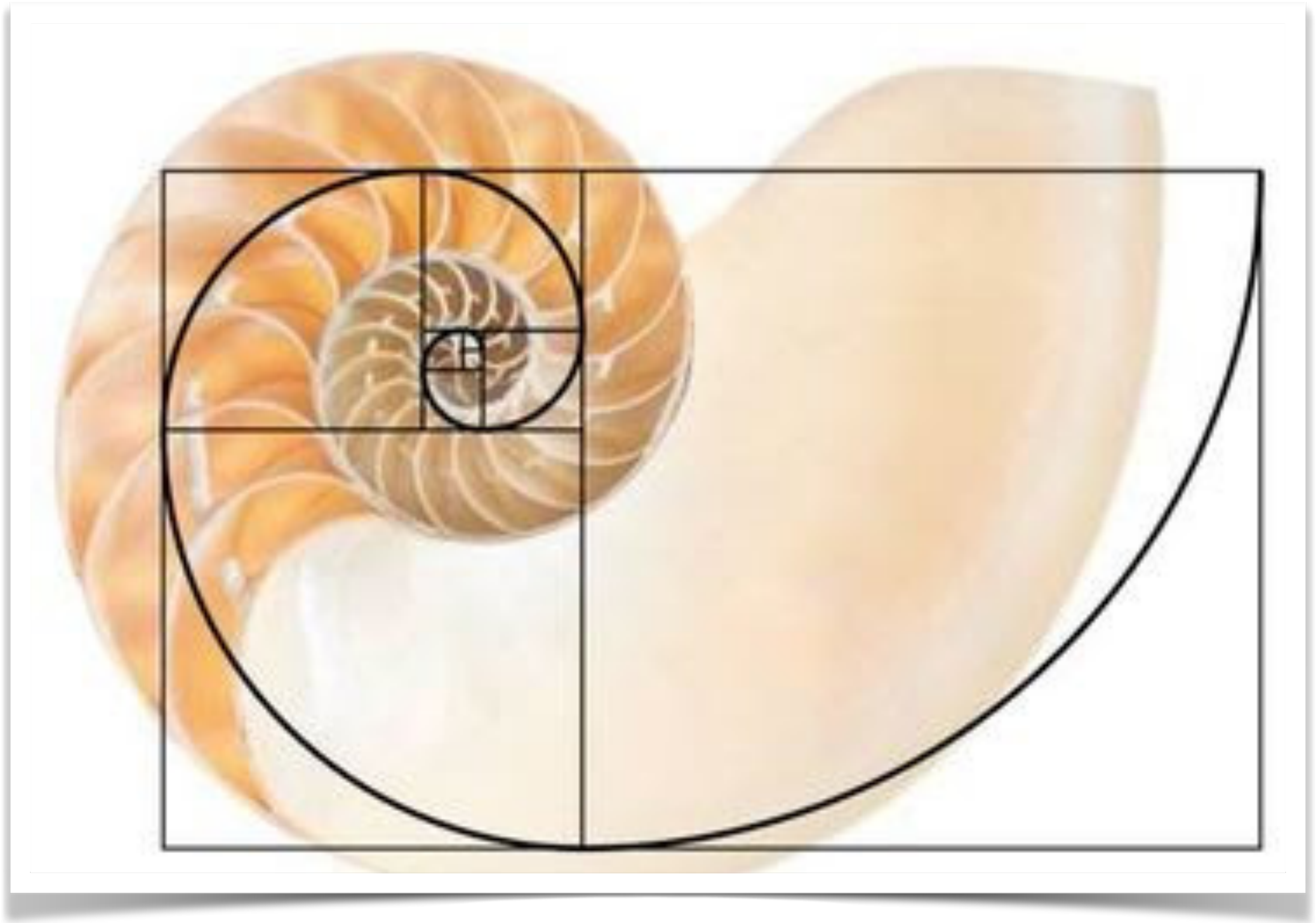
Architecture

Taj Mahal

Golden Spiral







Art

The Great Wave off Kanagawa

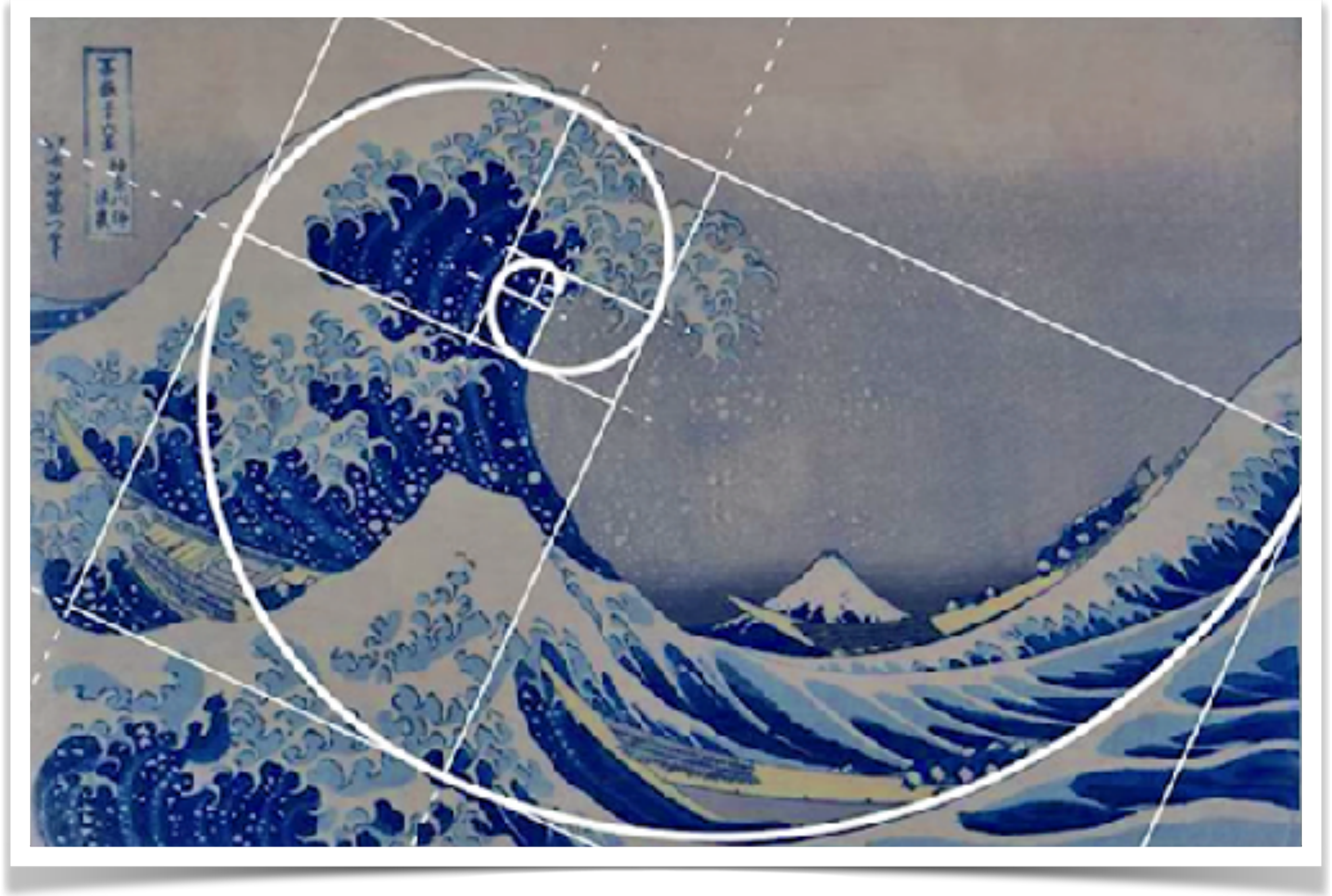
Architecture

Taj Mahal

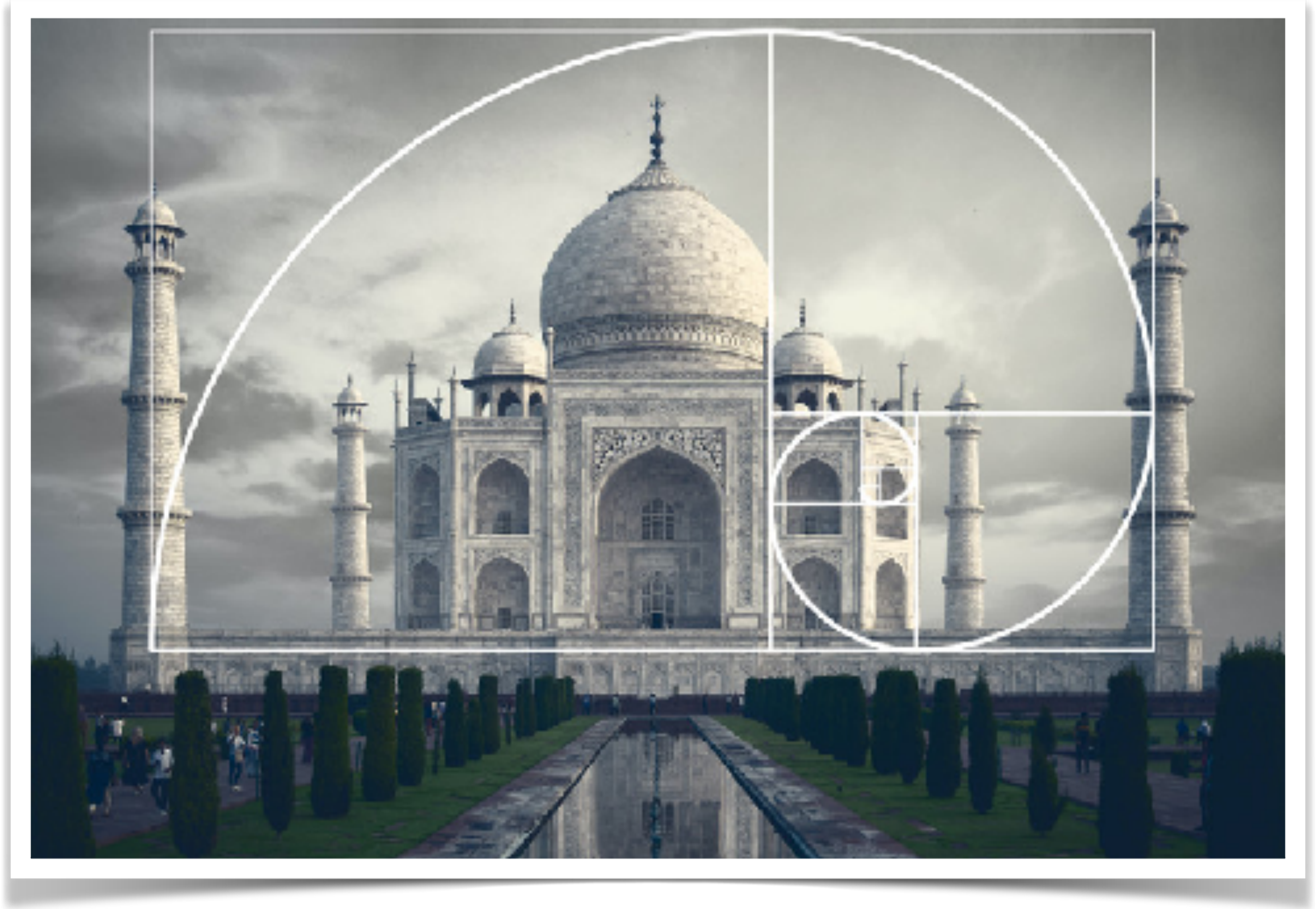
Nature

Nautilus Shell

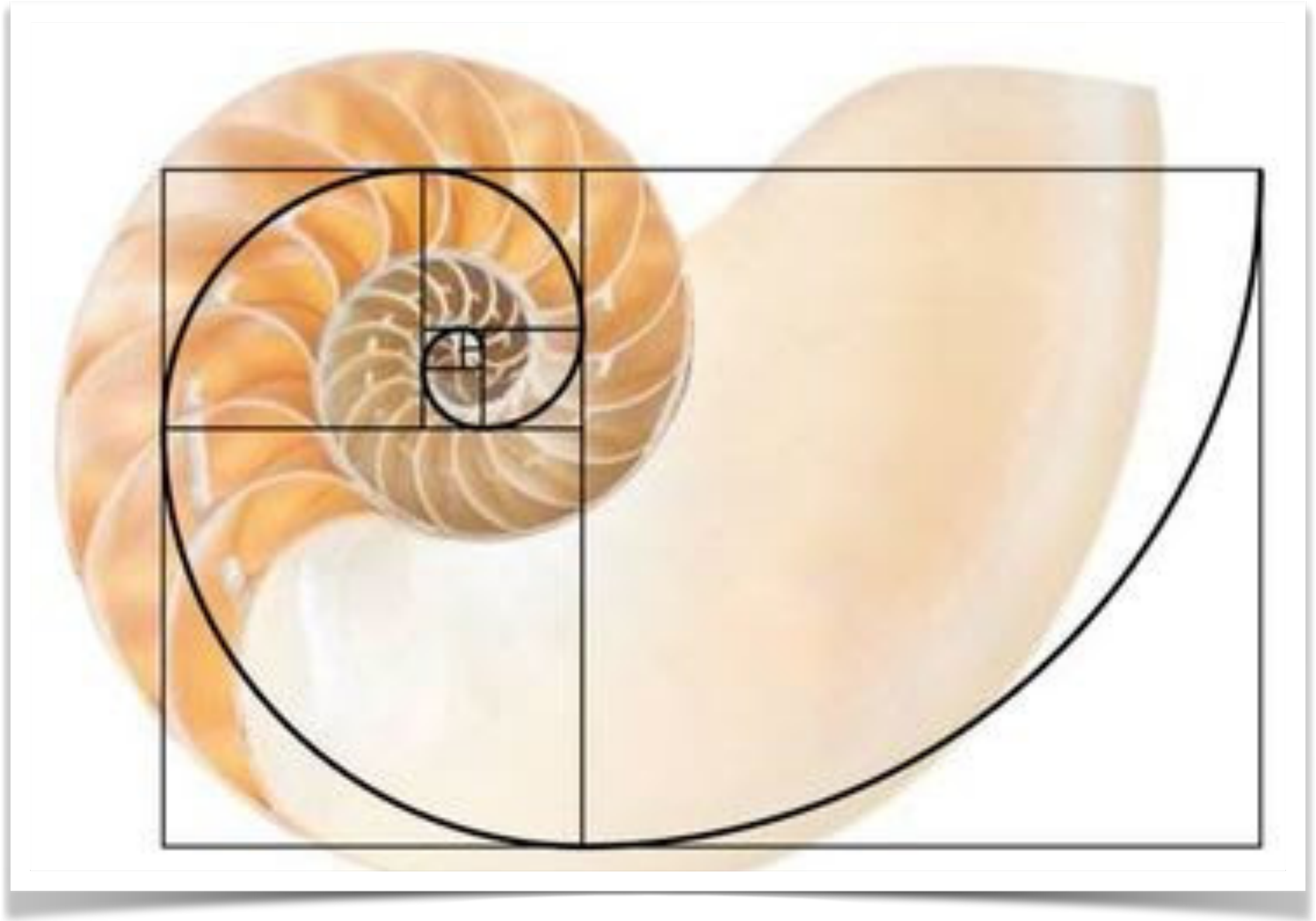
Golden Spiral



Art



Architecture



Nature

And now also in computer science :)

Weird Properties

$$\phi = \phi^2 - 1 = \frac{1}{\phi - 1}$$

Expand Array by Golden Ratio ($G = \phi = 1.61\ldots$)

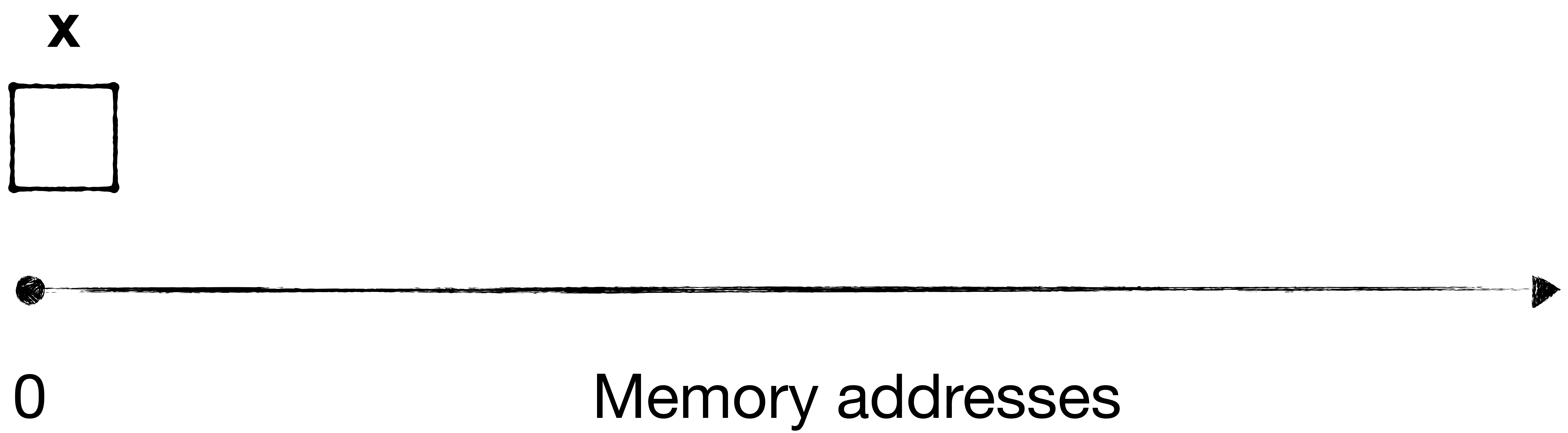
Expand Array by Golden Ratio (G = ϕ = 1.61…)

Satisfies both:

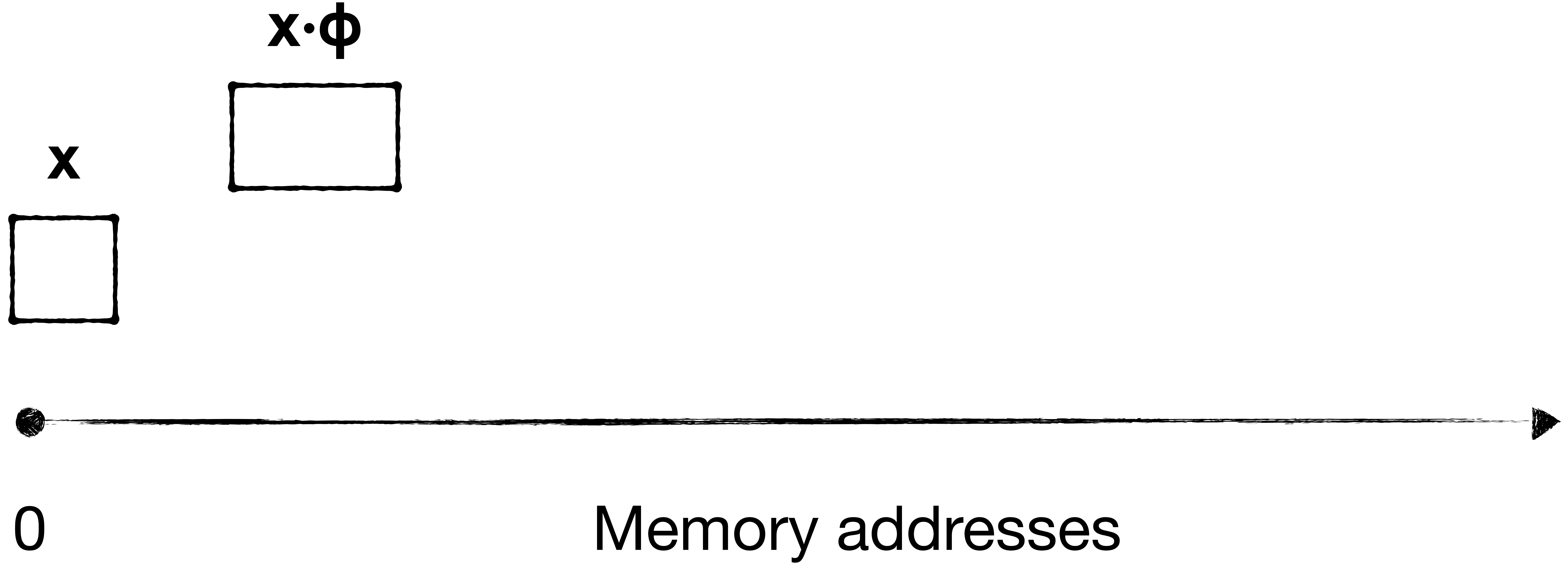
$$\text{Size}_{i-2} + \text{Size}_{i-1} = \text{Size}_{i}$$

$$\frac{\text{Size}_{i-1}}{\text{Size}_{i-2}} = \frac{\text{Size}_{i}}{\text{Size}_{i-1}} = G$$

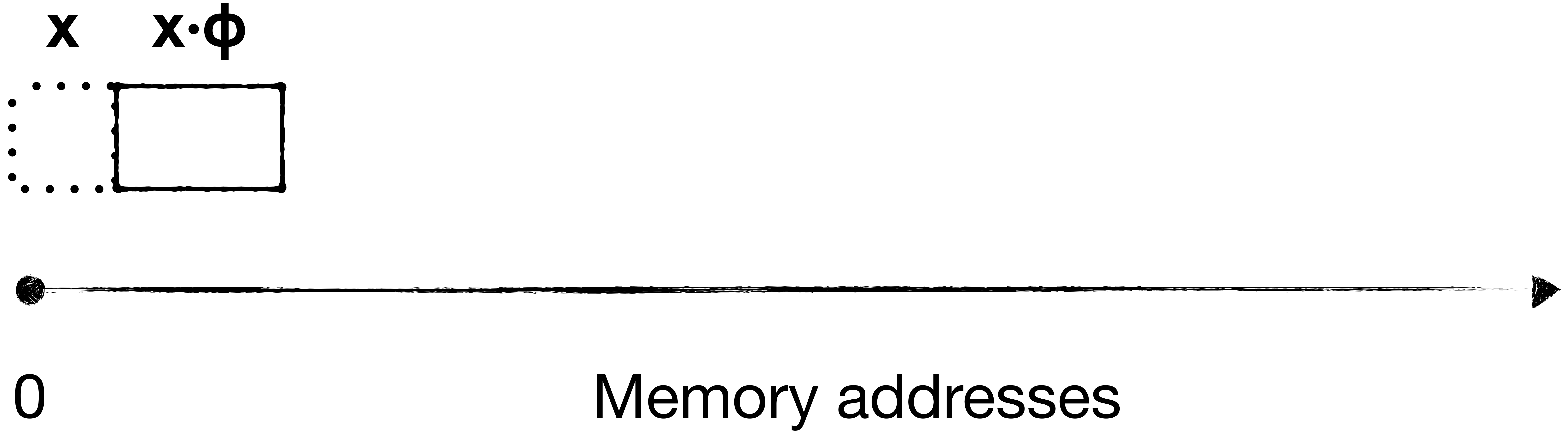
Expand Array by Golden Ratio (G = ϕ = 1.61…)



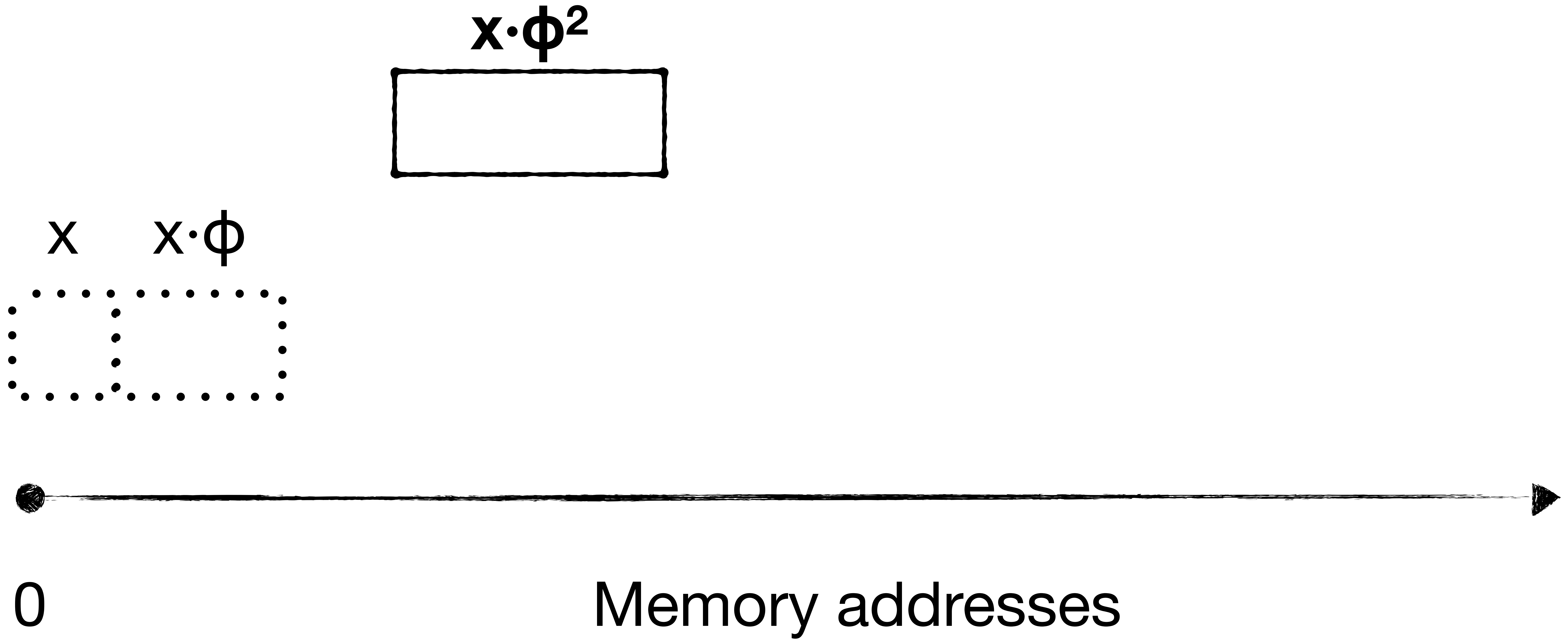
Expand Array by Golden Ratio (G = ϕ = 1.61…)



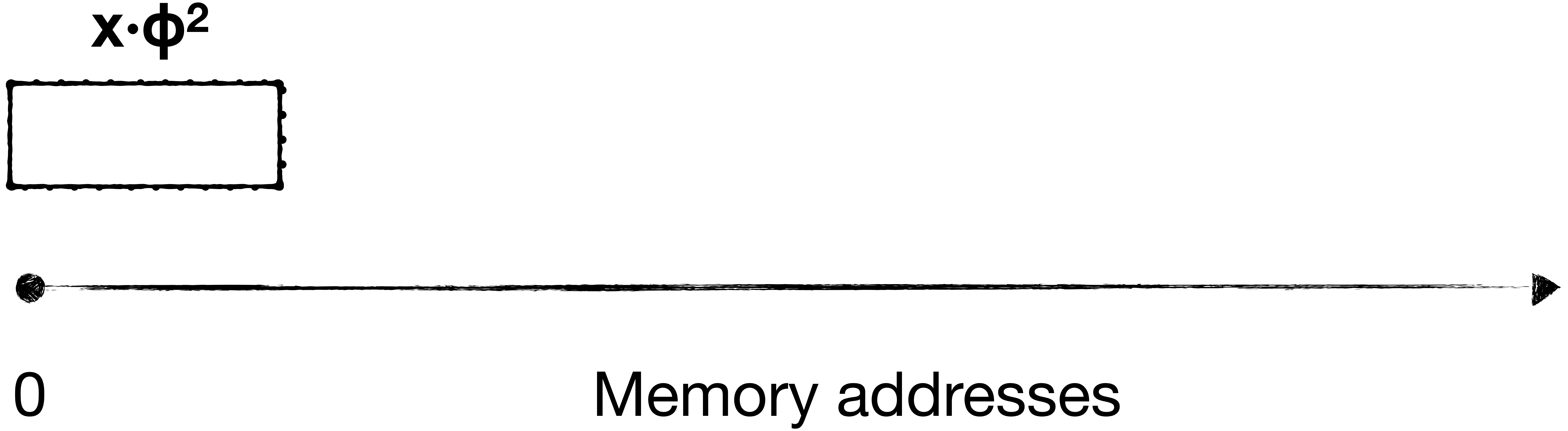
Expand Array by Golden Ratio (G = ϕ = 1.61…)



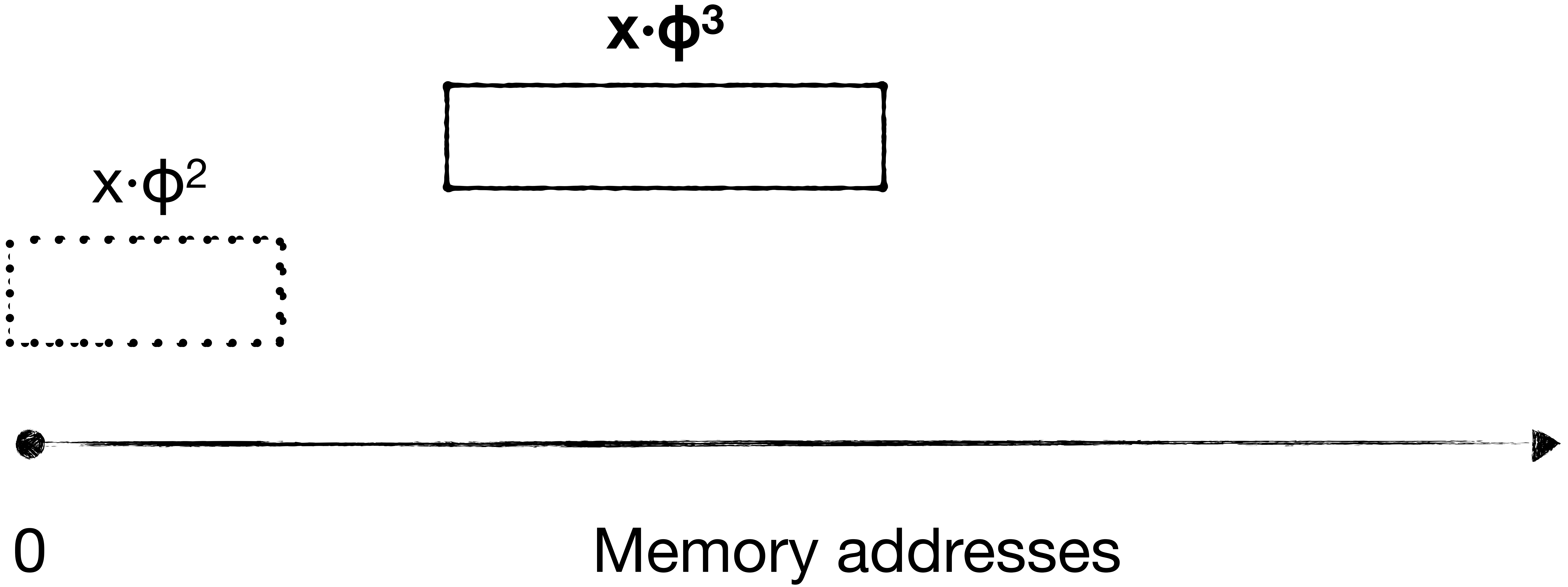
Expand Array by Golden Ratio (G = ϕ = 1.61…)



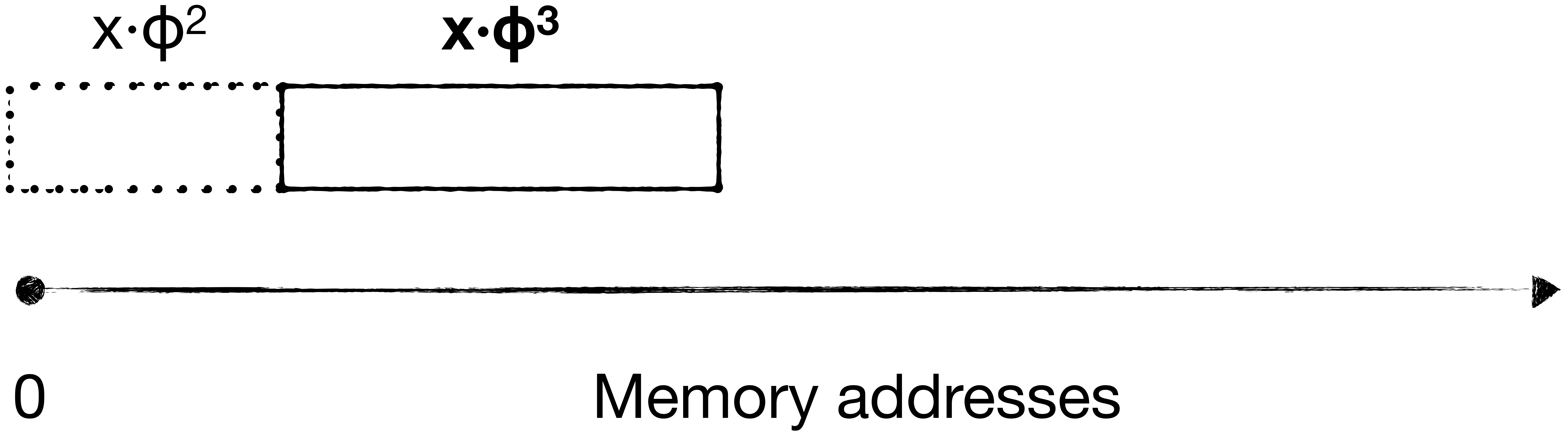
Expand Array by Golden Ratio (G = ϕ = 1.61…)



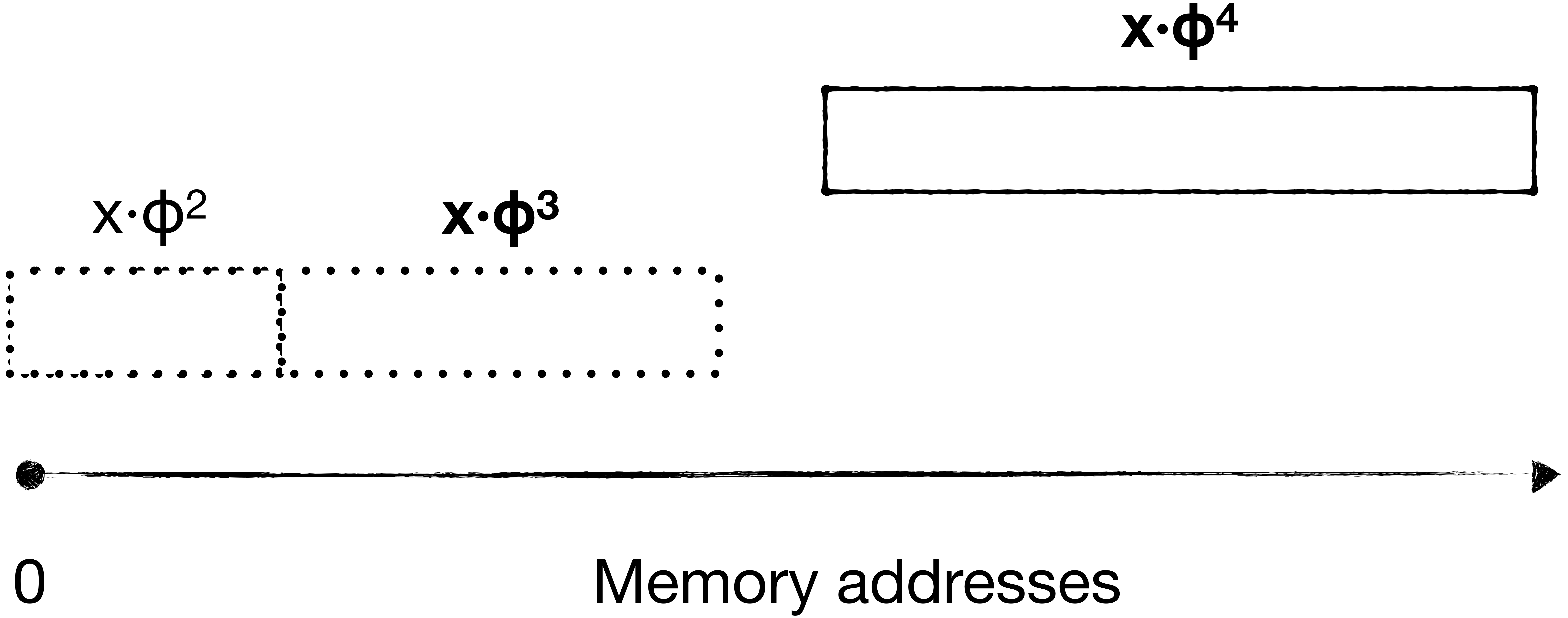
Expand Array by Golden Ratio (G = ϕ = 1.61…)



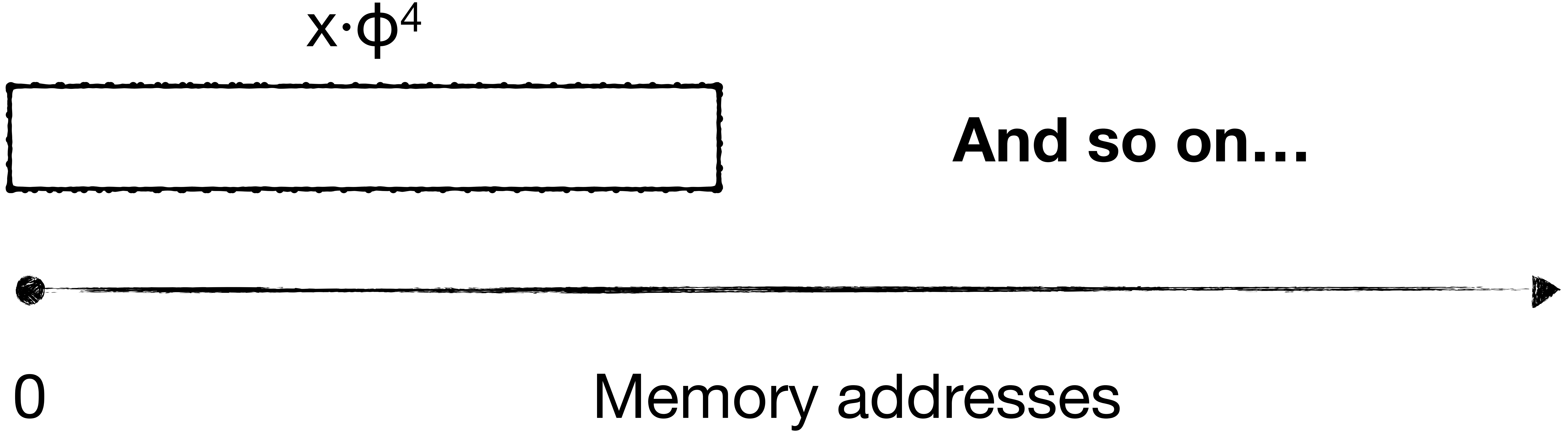
Expand Array by Golden Ratio ($G = \phi = 1.61\ldots$)



Expand Array by Golden Ratio (G = ϕ = 1.61…)

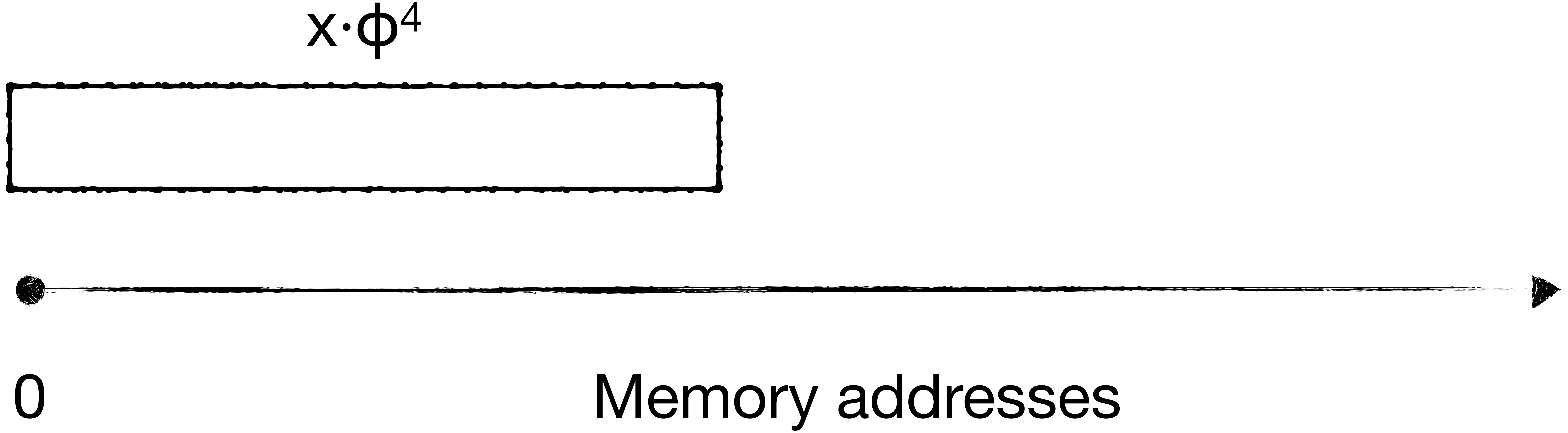


Expand Array by Golden Ratio (G = ϕ = 1.61…)

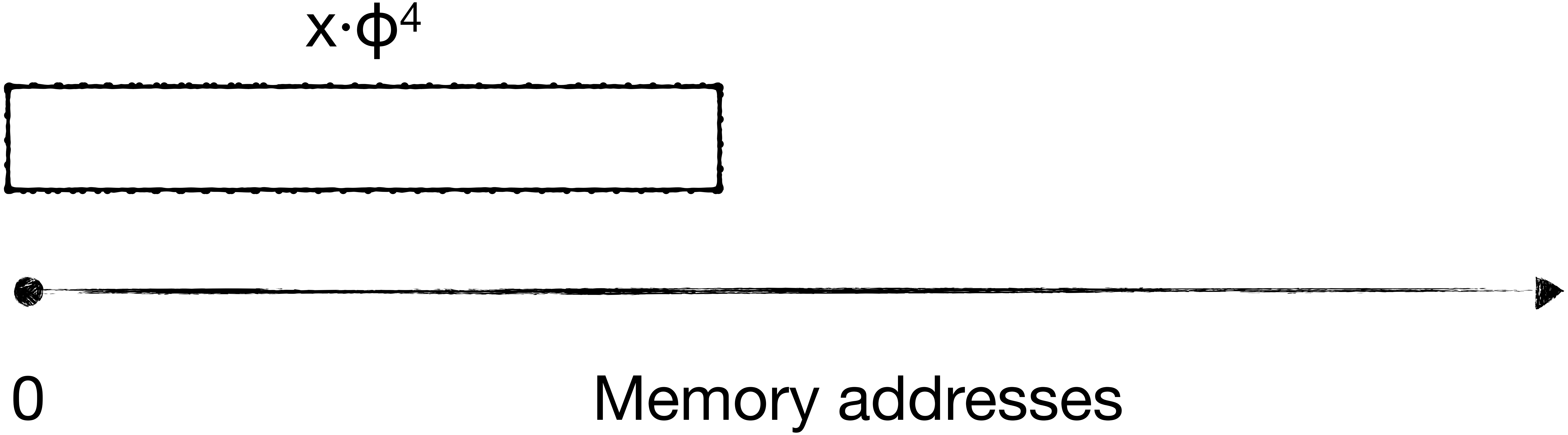


And so on…

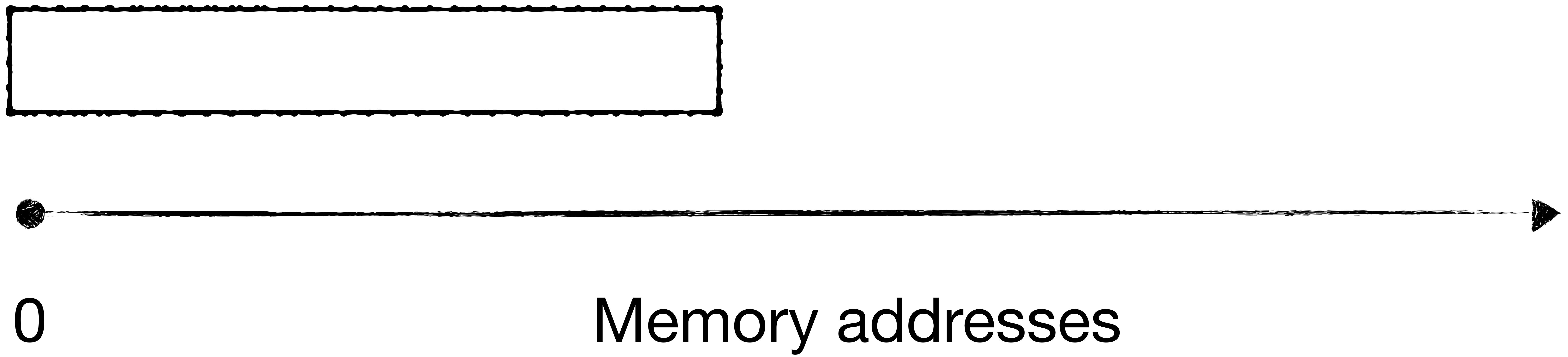
Write-Amplification



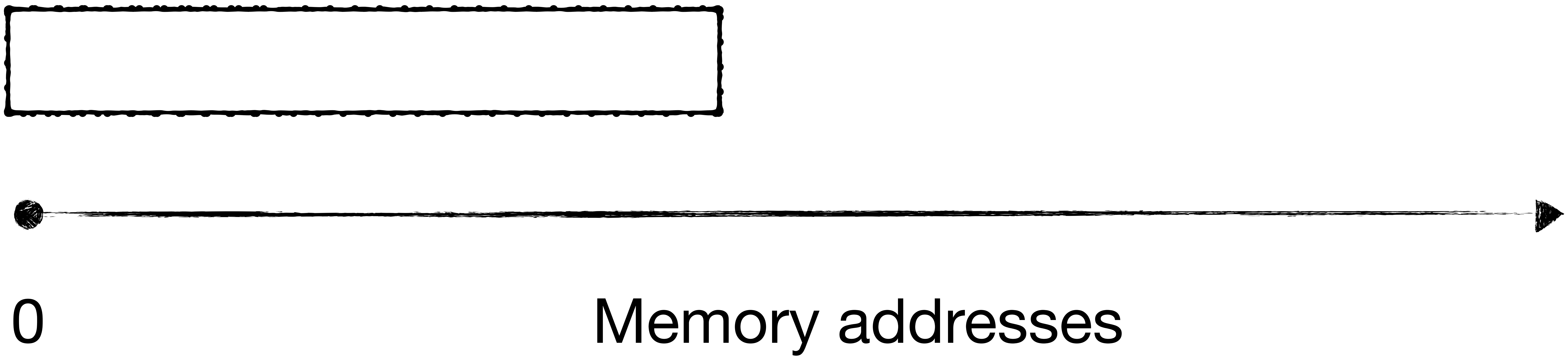
$$\textbf{Write-Amplification} = \frac{G}{G - 1}$$



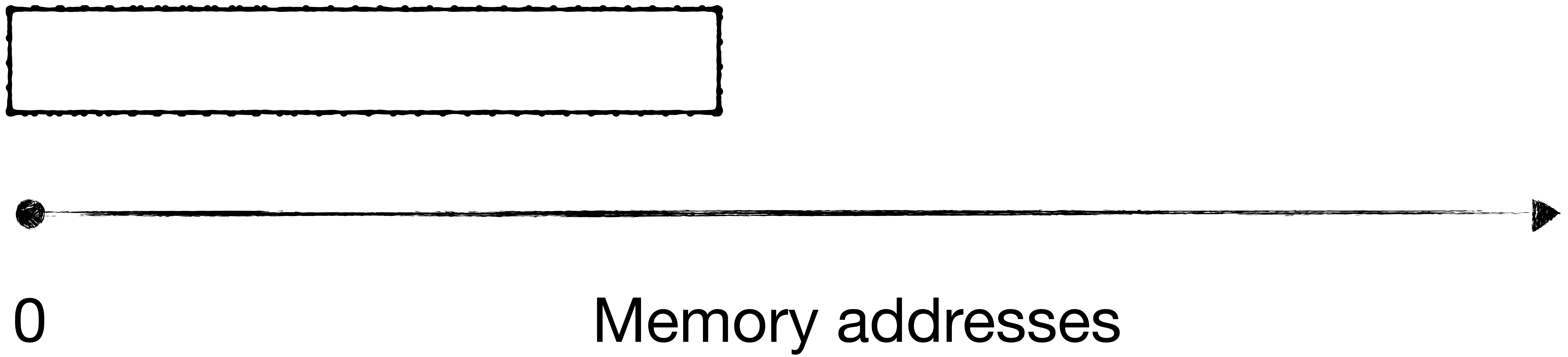
$$\textbf{Write-Amplification} = \frac{G}{G - 1} = \frac{\phi}{\phi - 1}$$



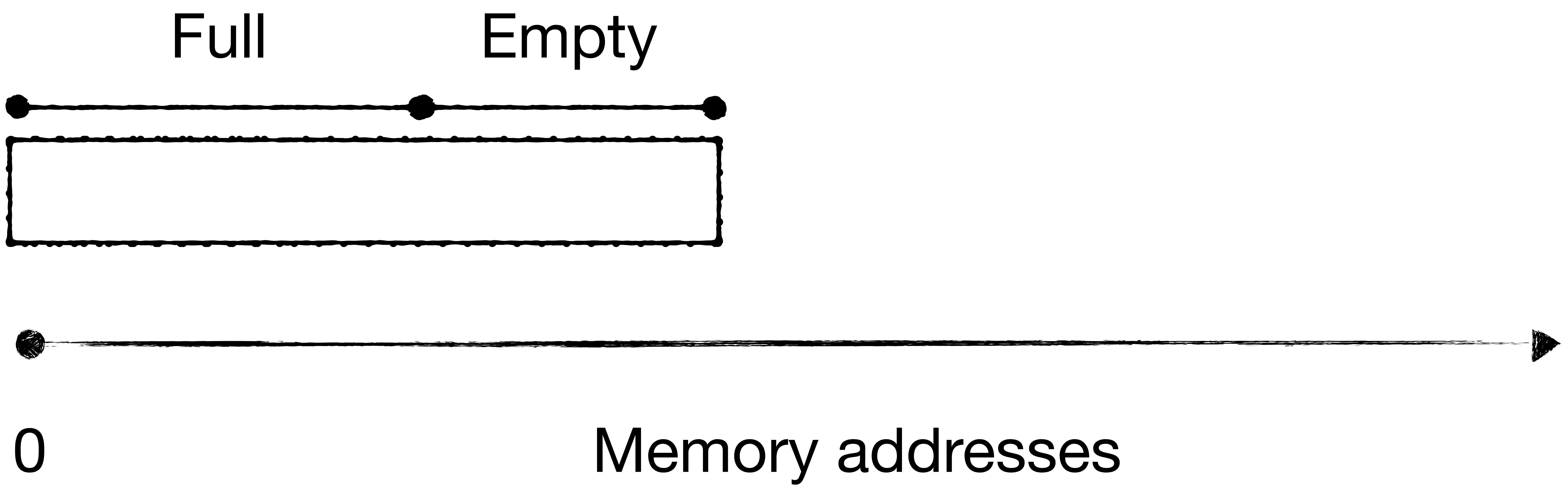
$$\textbf{Write-Amplification} = \frac{G}{G - 1} = \frac{\phi}{\phi - 1} = \phi + 1$$



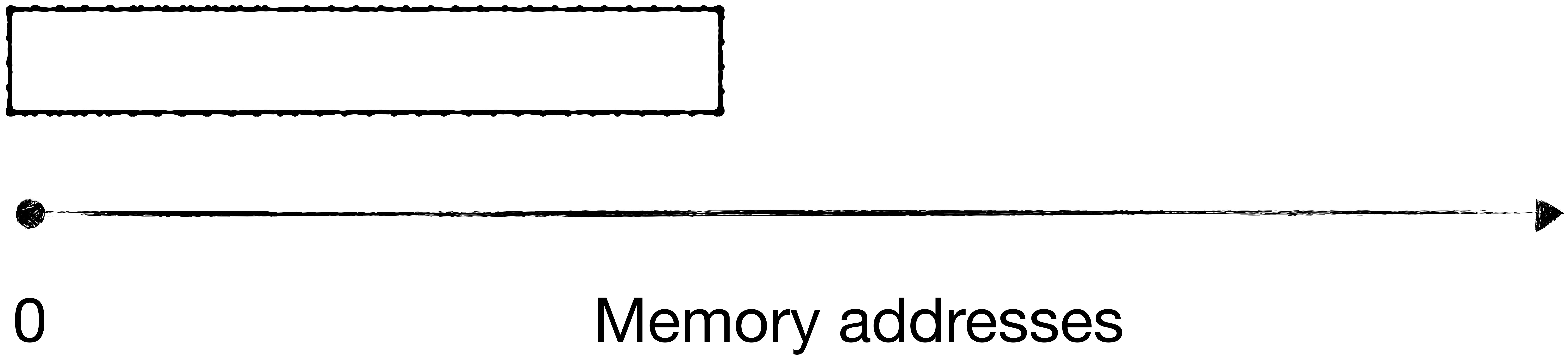
Space-Amplification?



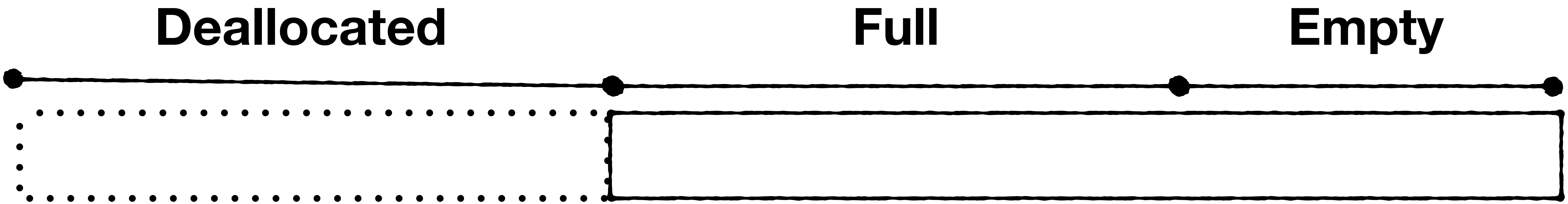
Space-Amplification?



$$\text{Space-Amplification} = \frac{\textbf{Full + Empty}}{\textbf{Full}} = \mathbf{G} = \boldsymbol{\phi}$$

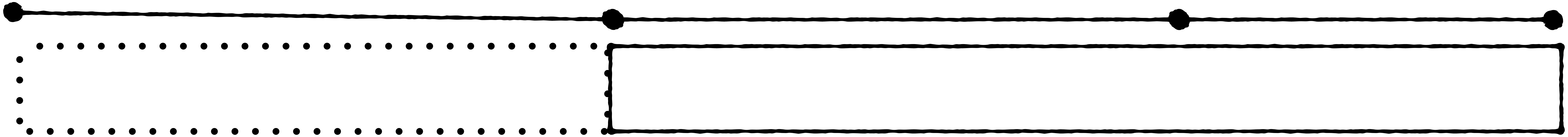


Max Space-Amp?

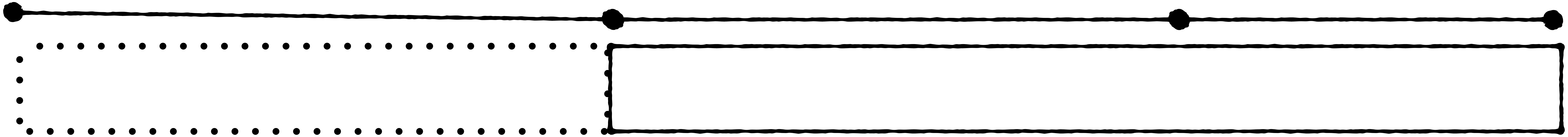


For G < ϕ

$$\text{Max Space-Amp} = \frac{\textbf{Deallocated + Full + Empty}}{\textbf{Full}} = \mathbf{1 + G}$$



Max Space-Amp $=$ **$1 + \phi$** $=$ **2.61**



	Write-amp	Space-amp
$G > \phi$	$\frac{G}{G - 1}$	$\frac{G}{G - 1} + G$
$G < \phi$	$\frac{G}{G - 1}$	Alternates G to G+1

In the wild

Implementation	Growth factor
Java ArrayList	1.5
Python PyListObject	~1.125
Microsoft Visual C++ 2013	1.5
G++ 5.2.0	2
Clang 3.6	2
Facebook folly/FBVector	1.5
Rust Vec	2
Go slices	between 1.25 and 2
Nim sequences	2
SBCL (Common Lisp) vectors	2
C# (.NET 8) List	2

Source: https://en.wikipedia.org/wiki/Dynamic_array#Growth_factor

Facebook folly/FBVector

https://github.com/facebook/folly/blob/main/folly/docs/FBVector.md

Real-world discussion of these issues

Facebook folly/FBVector

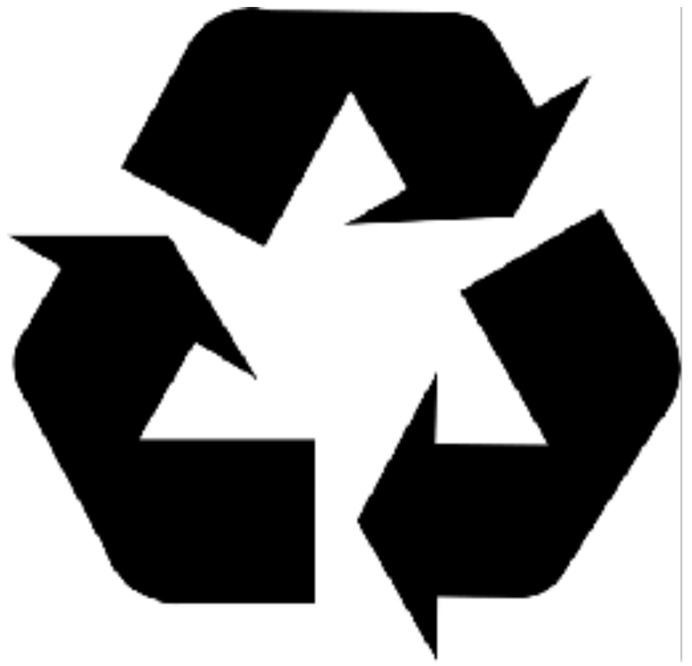
https://github.com/facebook/folly/blob/main/folly/docs/FBVector.md

Real-world discussion of these issues

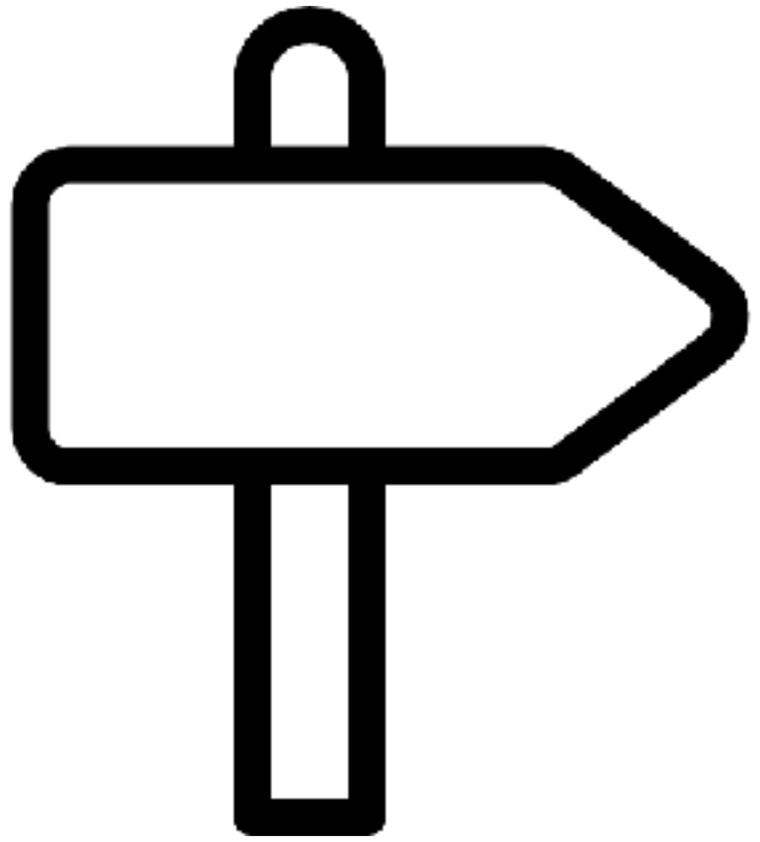
Note that Facebook also makes their own memory allocator, so with full control of the stack this can be more effective.

And now to new stuff

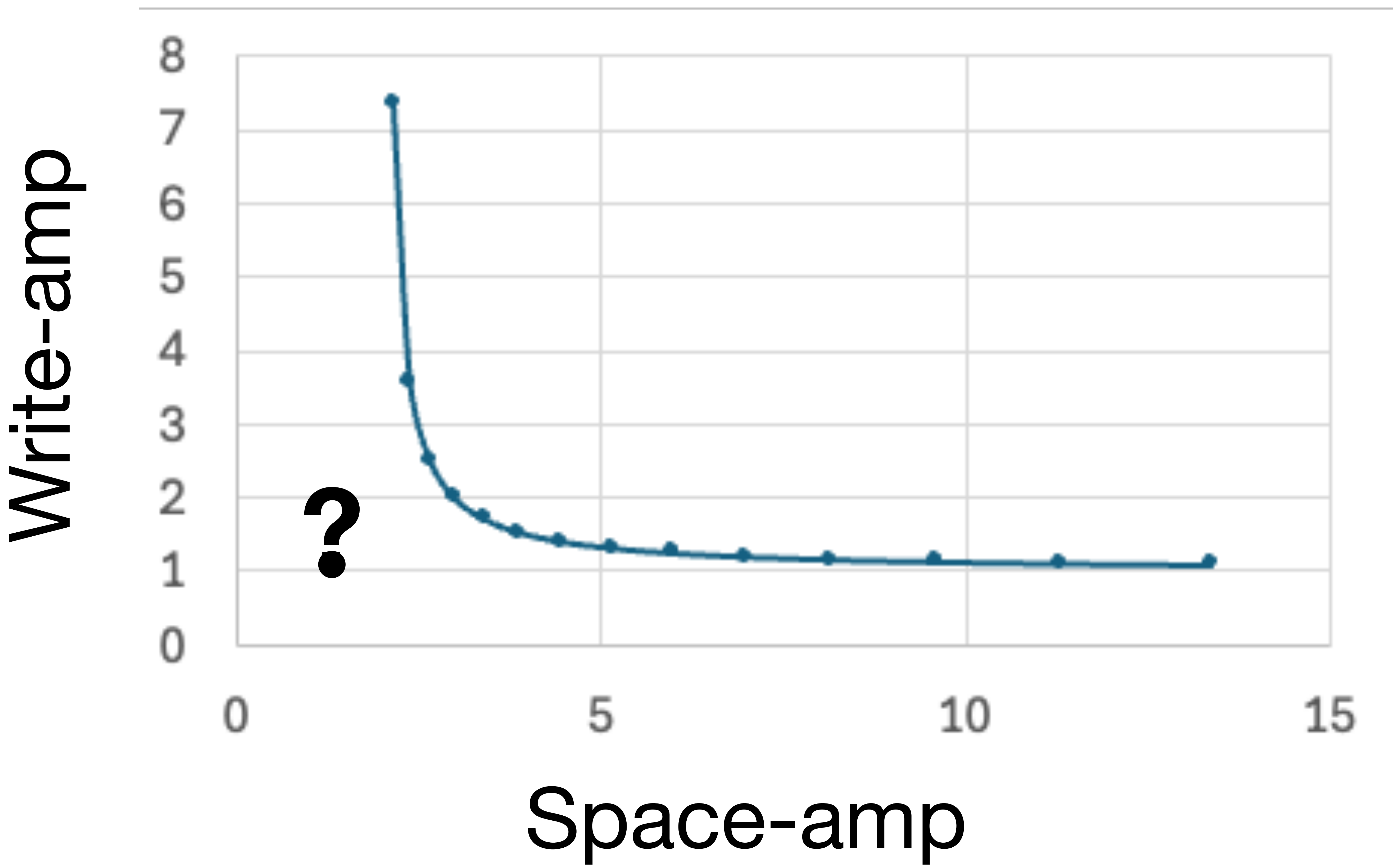
Reusing Deallocated Space



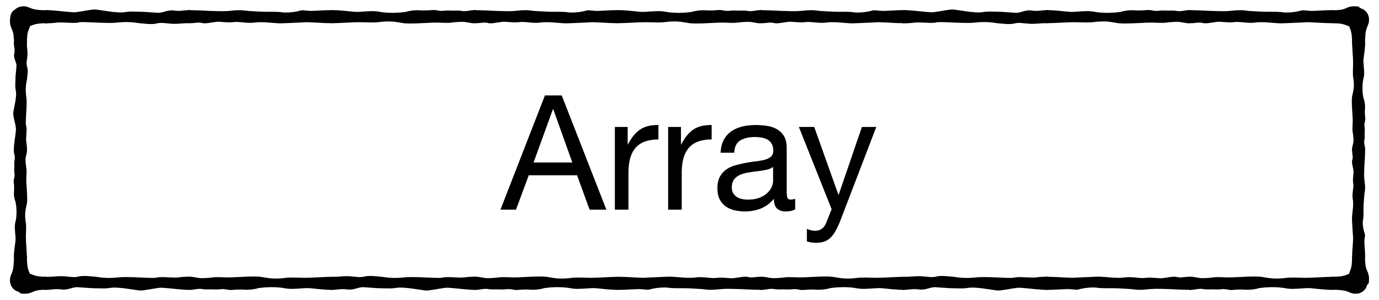
Alleviating trade-off via indirection



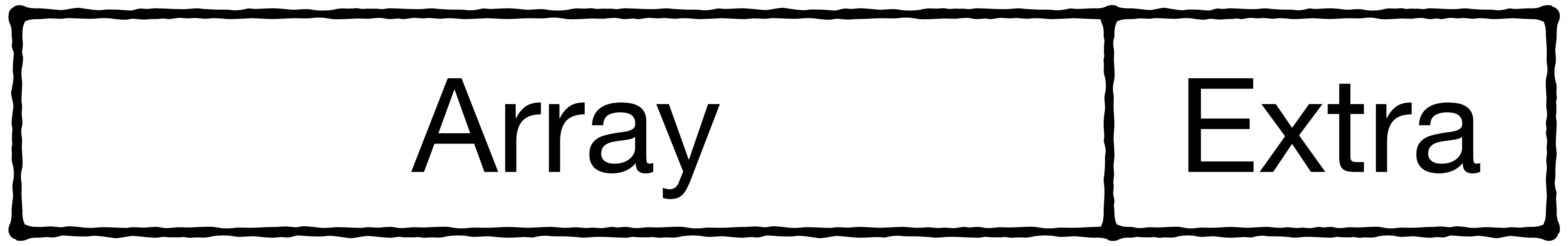
Can we completely overcome this trade-off?



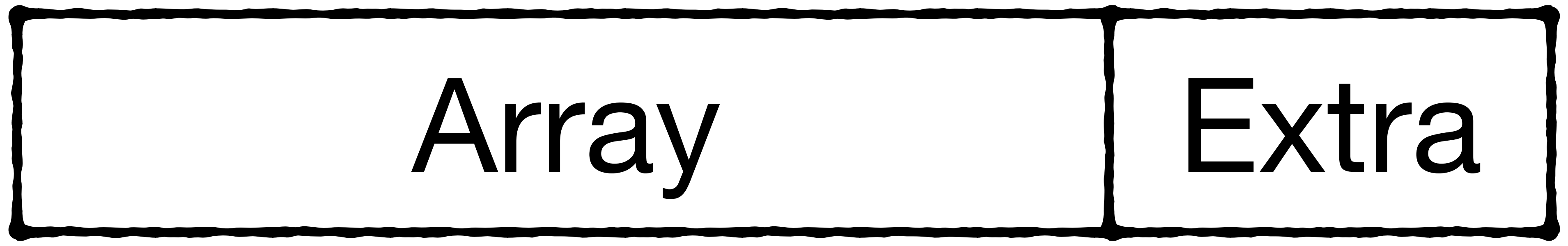
Suppose we could expand without copying everything:



Suppose we could expand without copying everything:



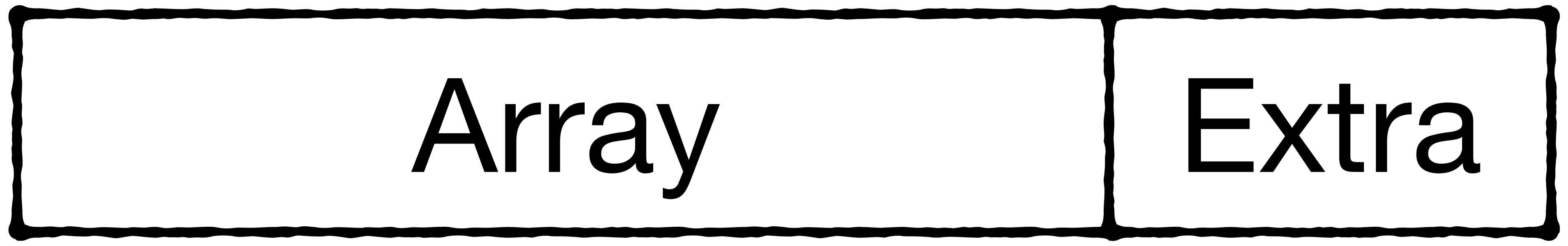
Suppose we could expand without copying everything:



Promise: **write-amp of ???**

space-amp of ???

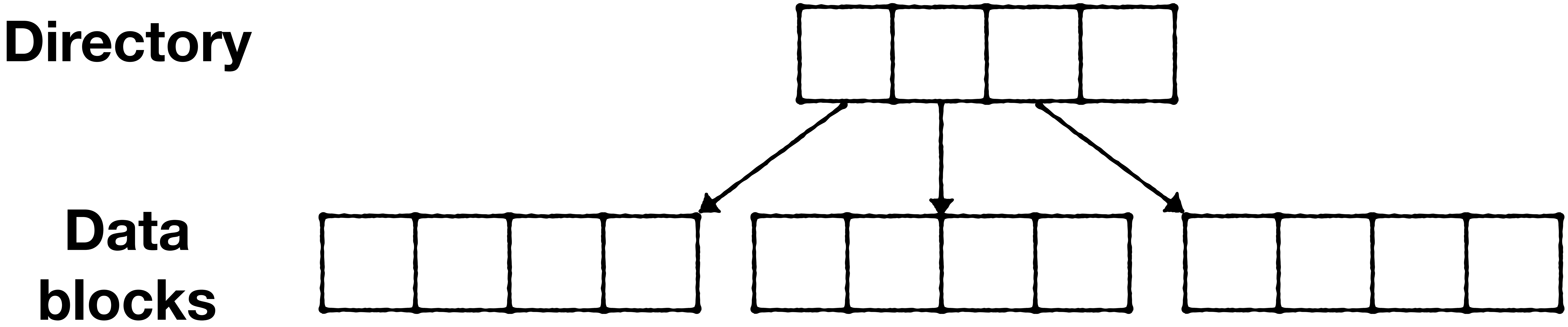
Suppose we could expand without copying everything:



Promise: **write-amp of ≈1**

space-amp of ≈1

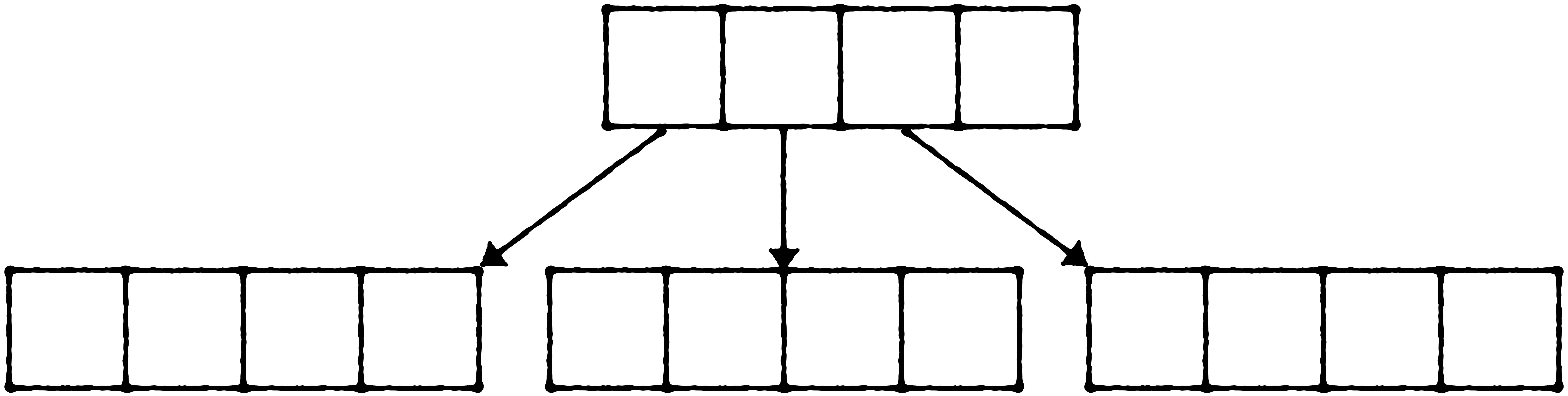
Add a layer of indirection



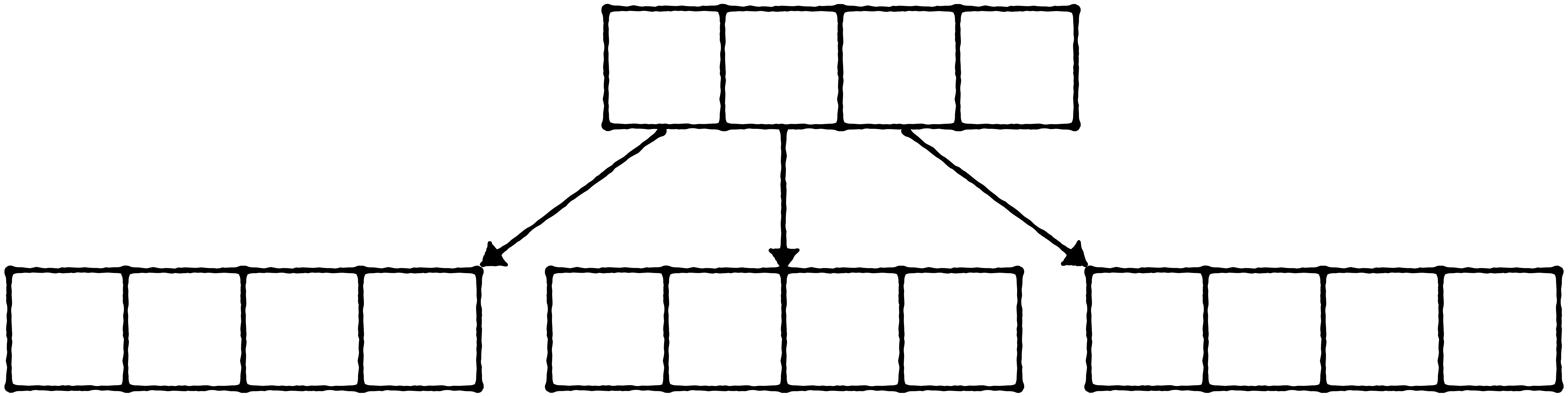
get(i)

Directory

Data
blocks



get(i) **Data block = ⌊i / data block size⌋**



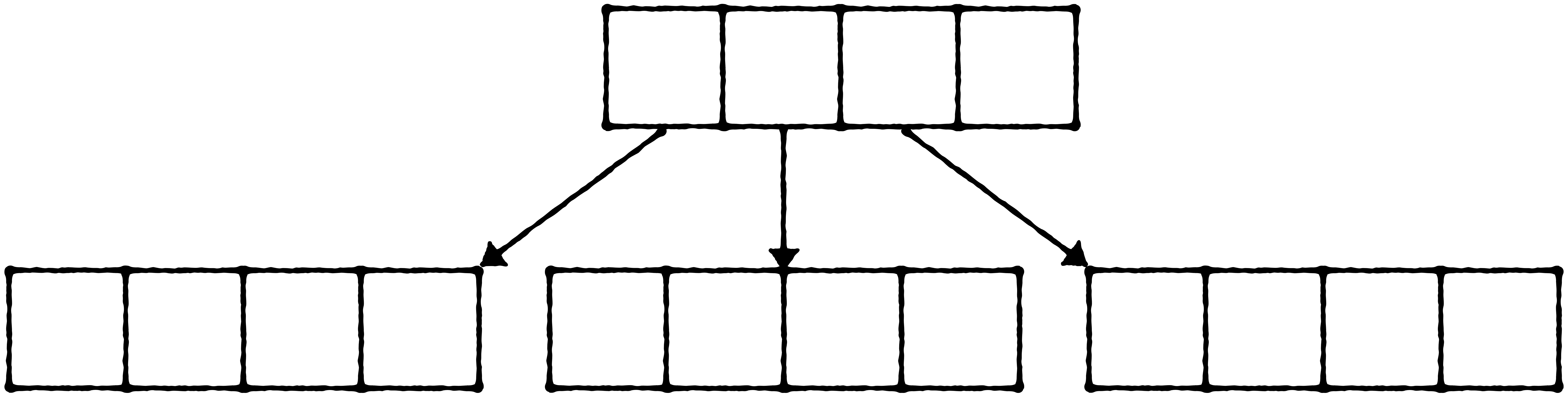
get(i)

Data block = ⌊ i / data block size ⌋

offset within = i % data block size

Directory

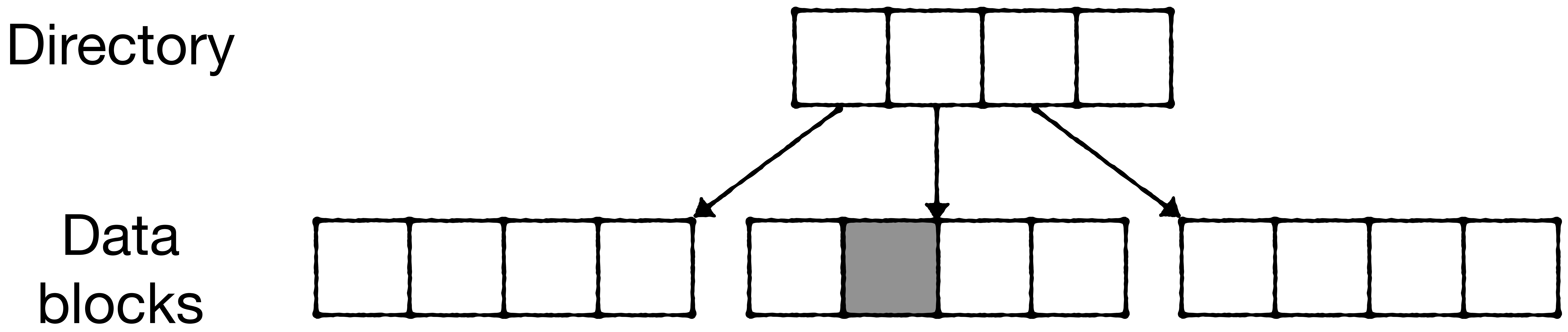
Data blocks



get(5)

Data block = ⌊5 / 4⌋ = **1**

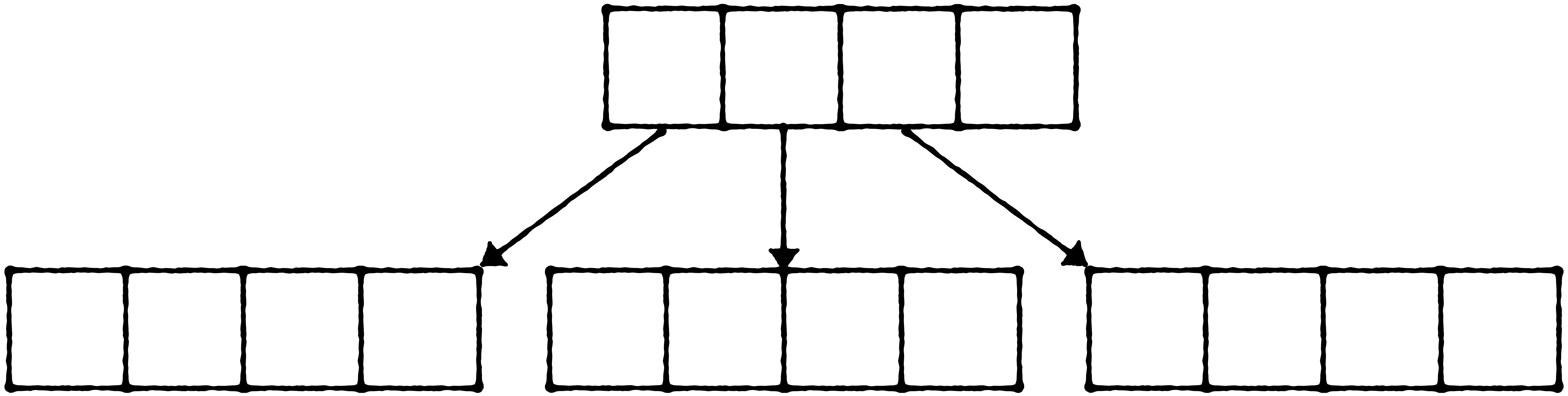
offset within = 5 % 4 = **1**



Expand?

Directory

Data
blocks



Expand?

Add new data block to directory

Expand?

Expand directory if we need more space

Expand?

Expand directory if we need more space

Downside?

Downside**: 2 memory hops per access**

Downside: 2 memory hops per access

Mitigation?

Downside: 2 memory hops per access

Mitigation: directory must fit in L1 cache

Downside: 2 memory hops per access

Mitigation: directory must fit in L1 cache

Typical L1 cache size:
16-128 KB per core

directory size?

$$\text{directory size} = \frac{\text{Data size}}{\text{Data block size}}$$

$$\text{directory size} = \frac{\text{Data size}}{\text{Data block size}} = O(N)$$

Risk: data blocks are initialized too small

$$\text{directory size} = \frac{\text{Data size}}{\text{Data block size}} = O(N)$$

Risk: data blocks are initialized too small

Directory may outgrow the L1 cache

$$\text{directory size} = \frac{\text{Data size}}{\text{Data block size}} = O(N)$$

Risk: data blocks are initialized too small

Directory may outgrow the L1 cache

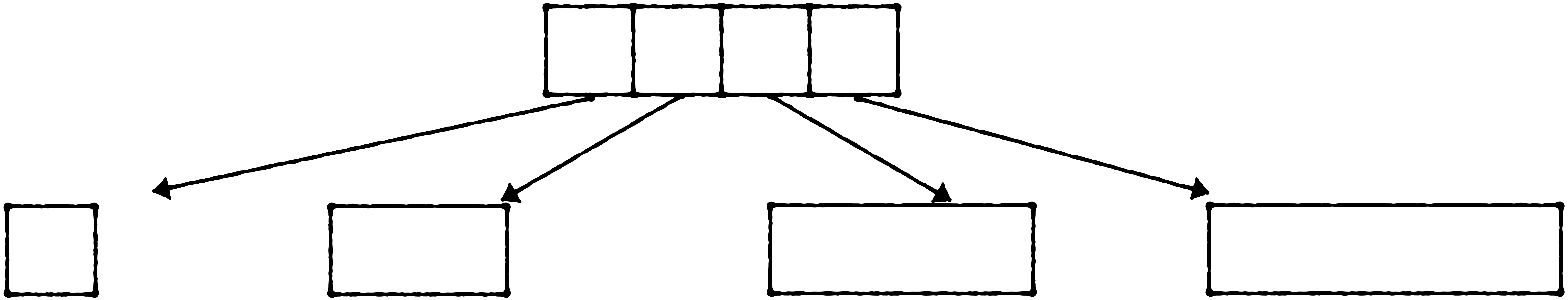
Solution?

Resizable Arrays in Optimal Time and Space

Algorithms and Data Structures Symposium, 1999

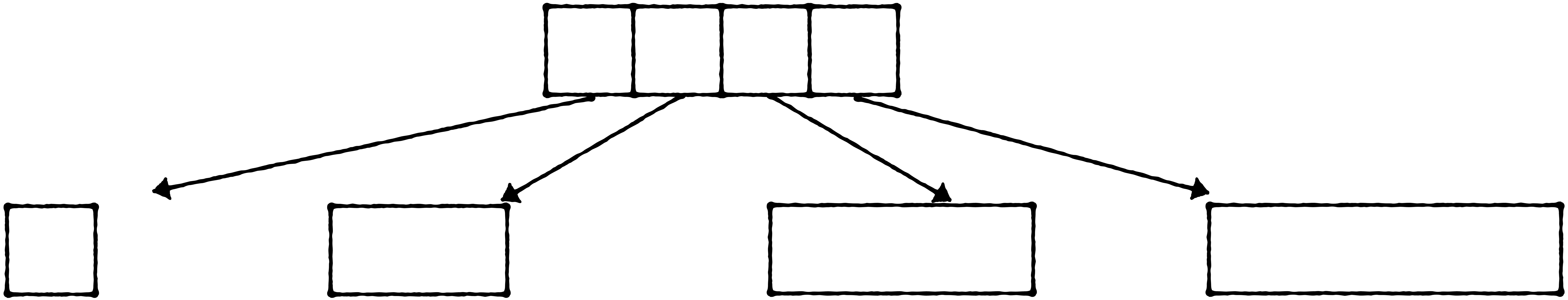
Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick

Resizable Arrays in Optimal Time and Space

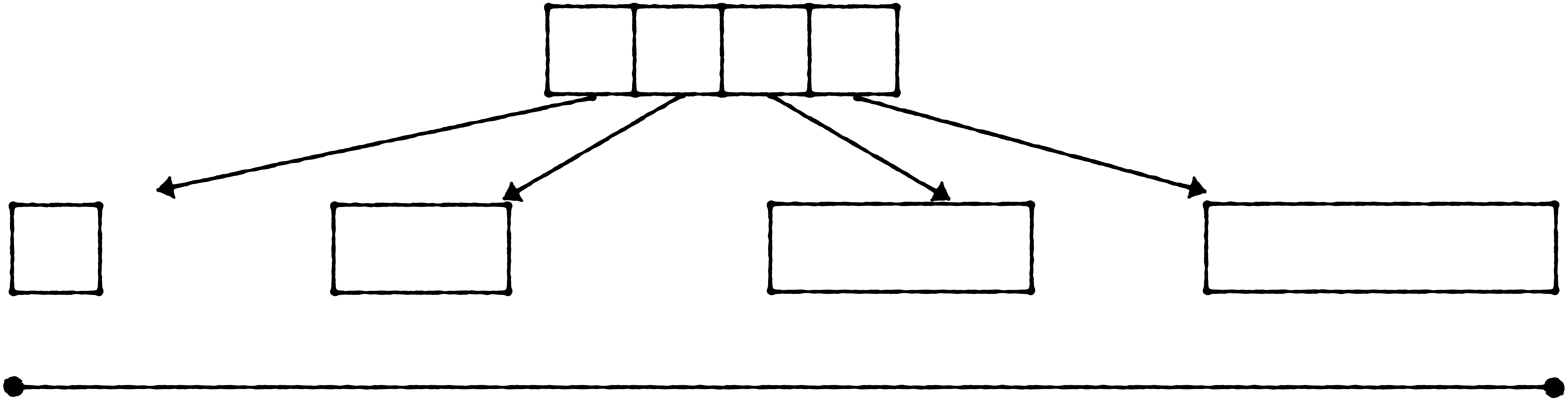


Data blocks should grow in size

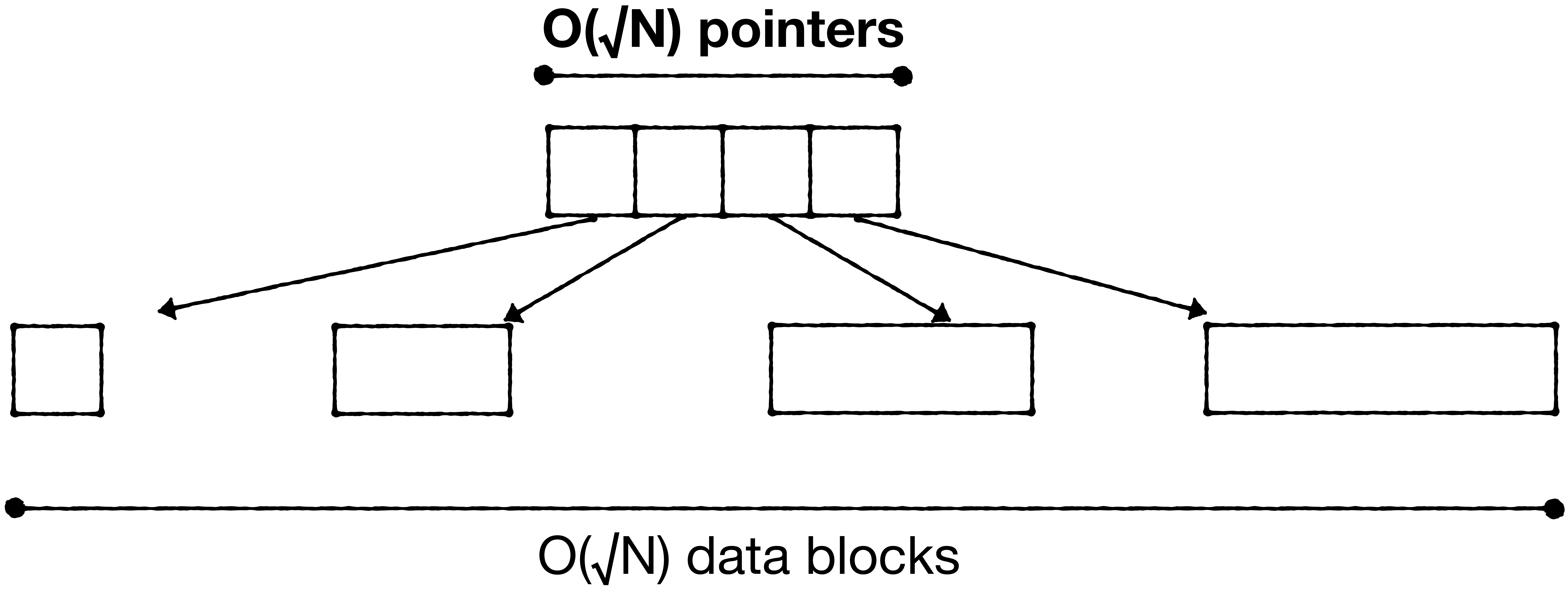
Resizable Arrays in Optimal Time and Space

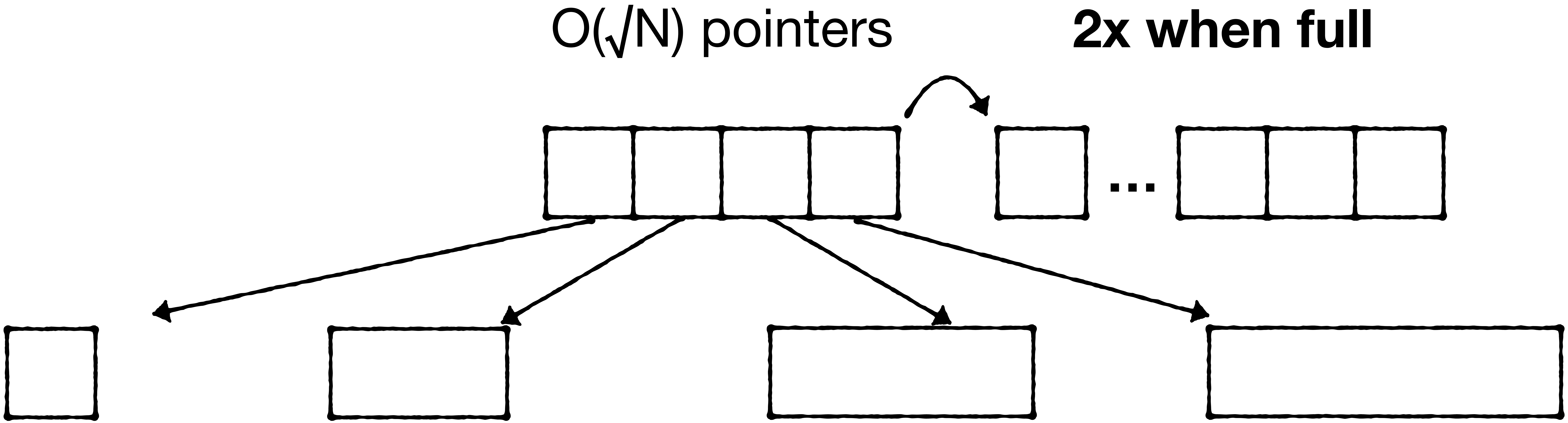


Data blocks should grow in size → **Directory grows more slowly**

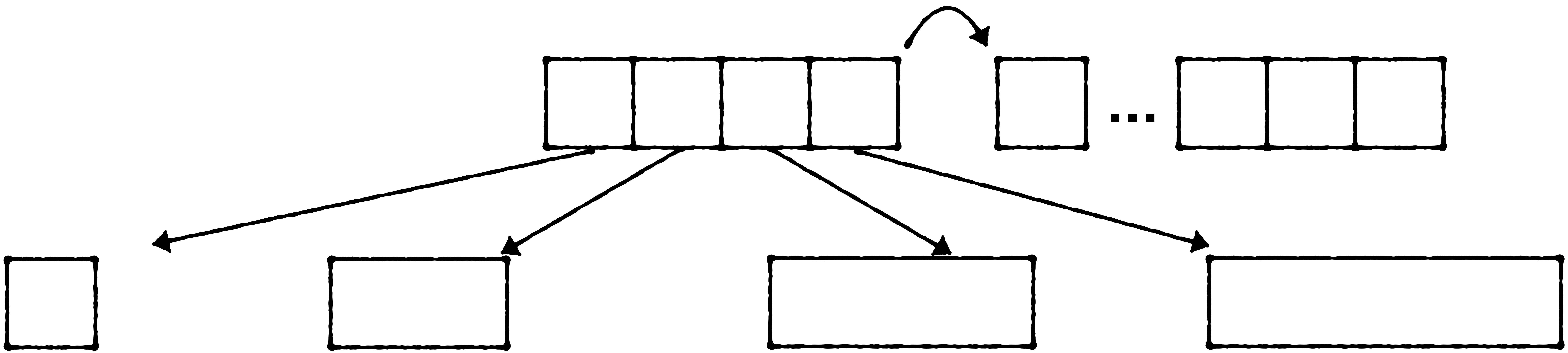


O(√N) data blocks

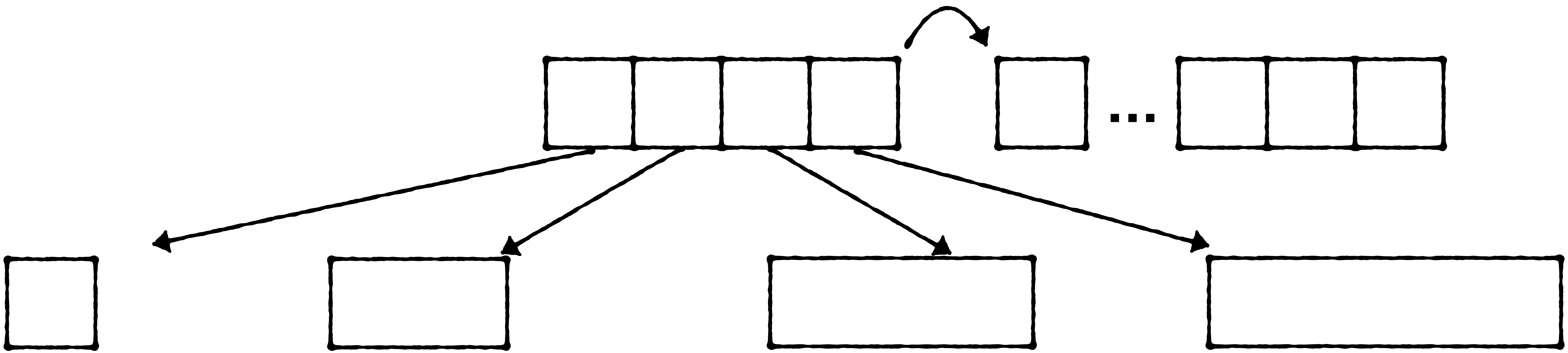

O(√N) pointers
O(√N) data blocks

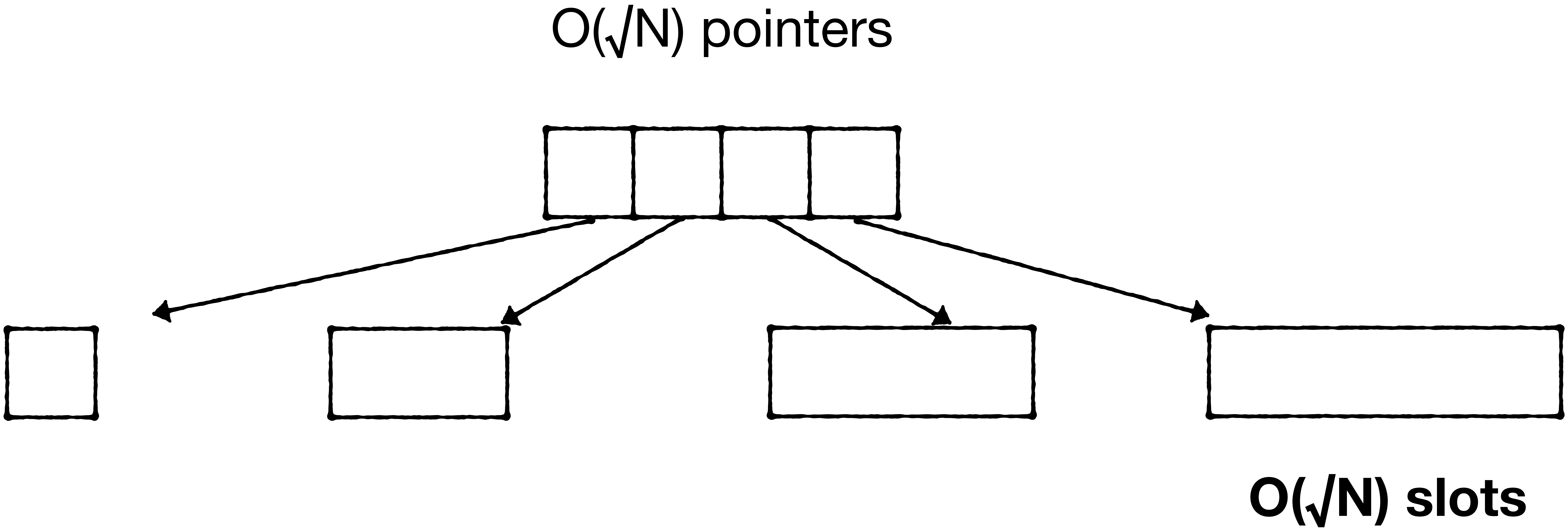

O(√N) pointers
2x when full
...

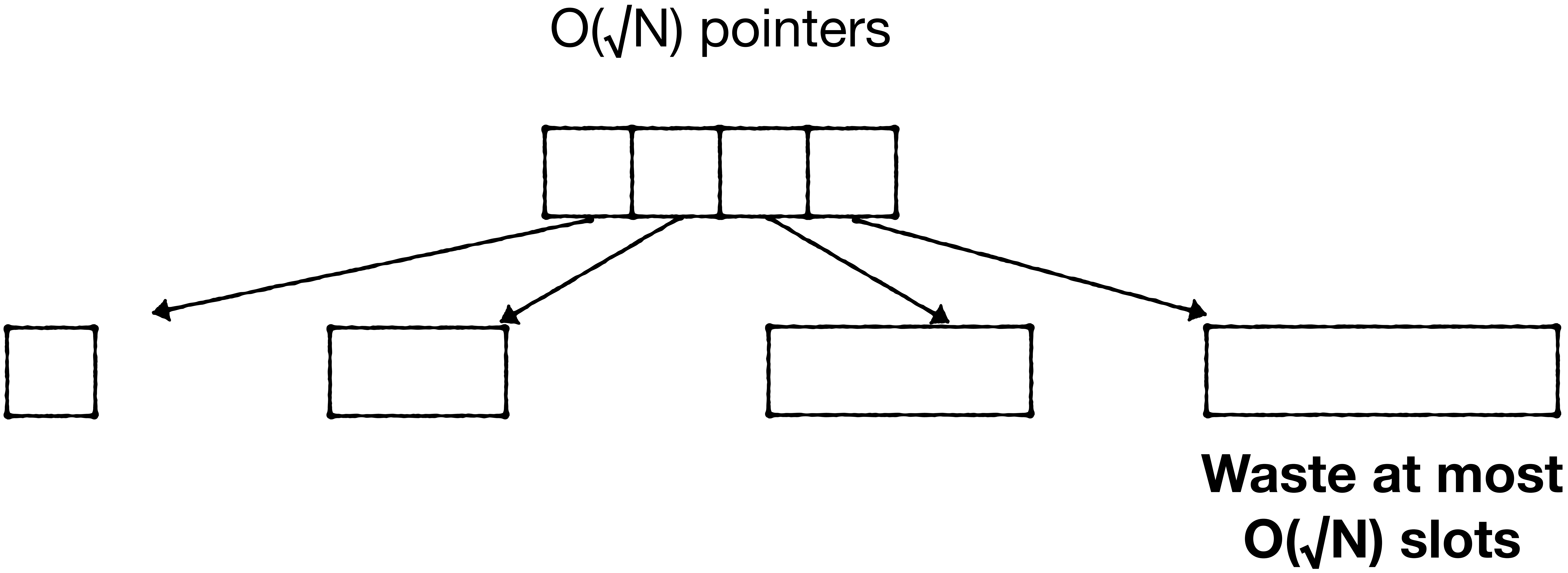
O(2√N) pointers



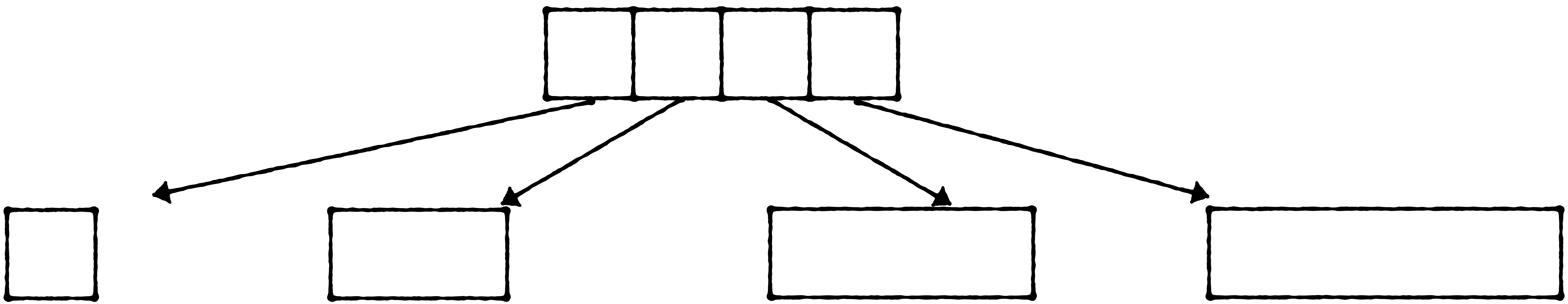
O(√N) pointers



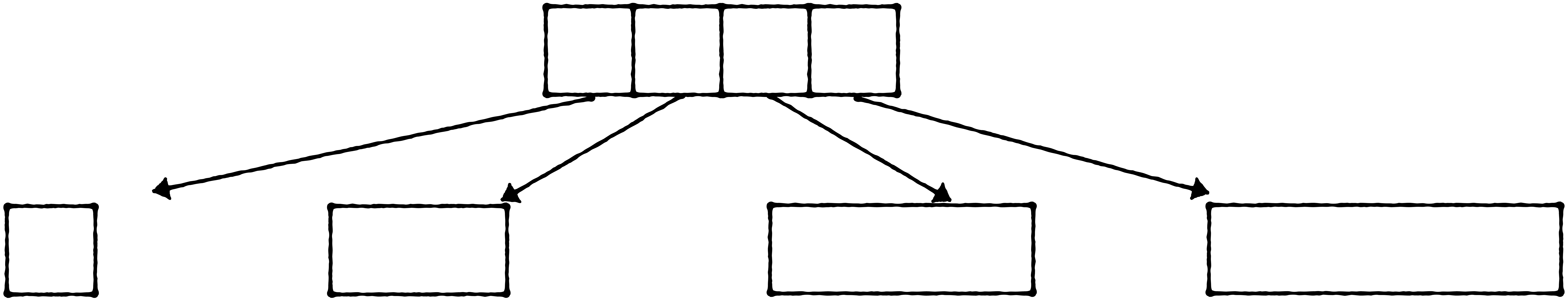

O(√N) pointers
O(√N) slots


O(√N) pointers
Waste at most
O(√N) slots

Max space amp = $O(\sqrt{N}) + O(\sqrt{N})$ **=** $\mathbf{O(\sqrt{N})}$

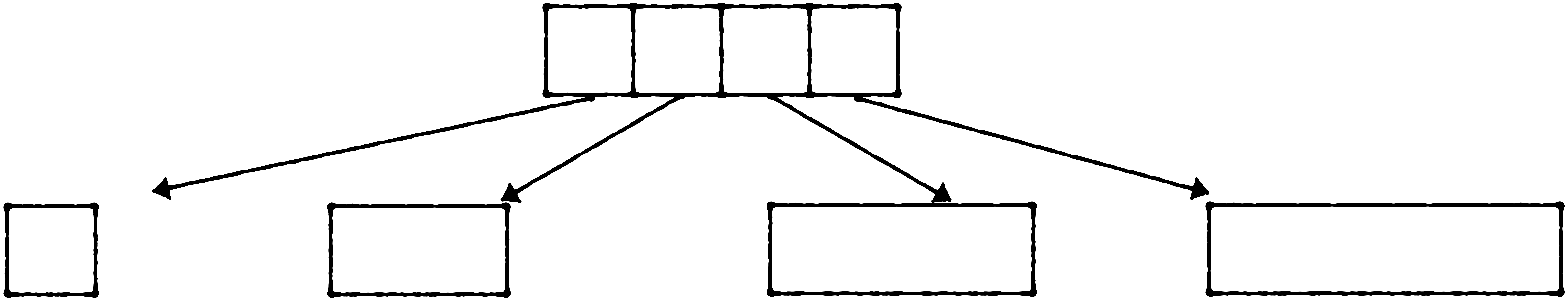


Max space amp $= O(\sqrt{N})$



Challenges: How to grow blocks to meet these properties?

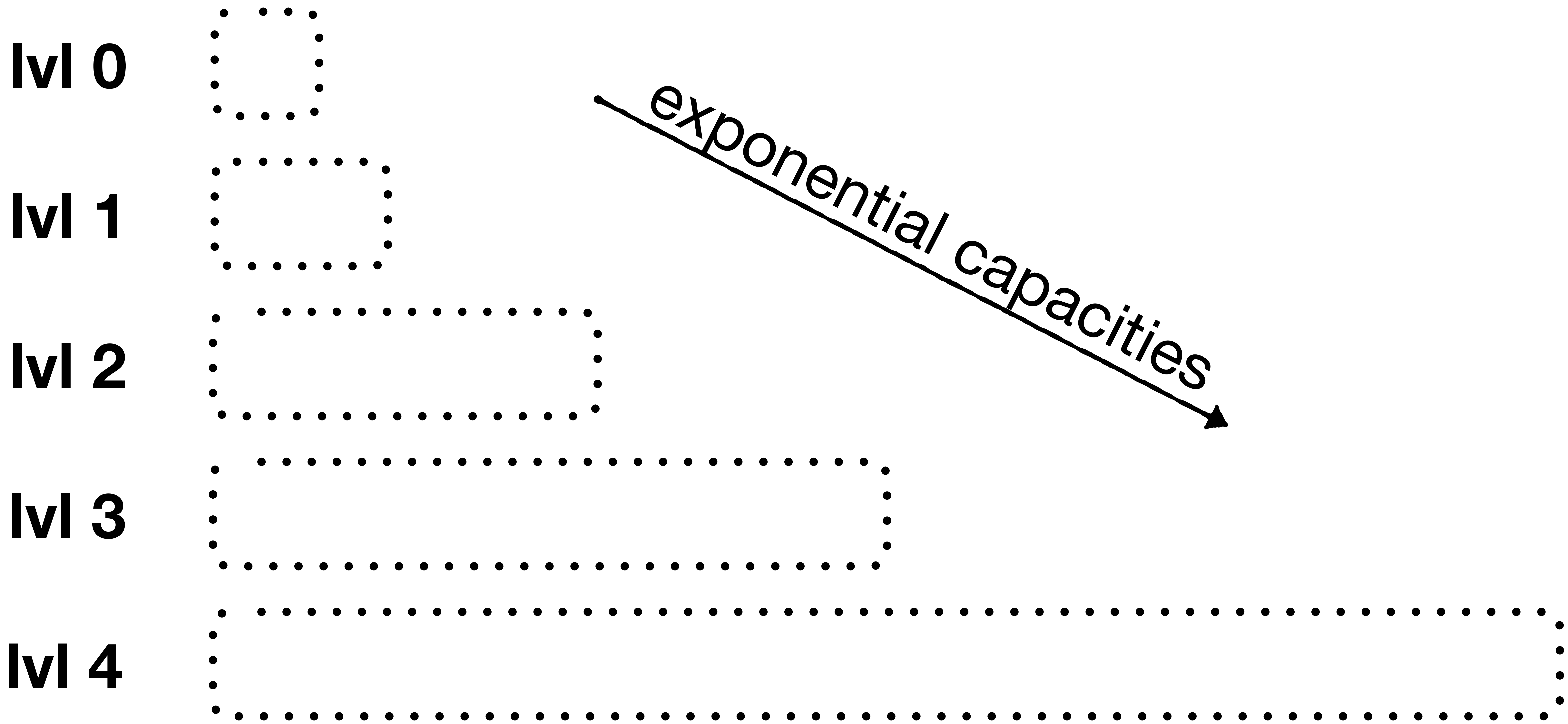
Max space amp = $O(\sqrt{N})$

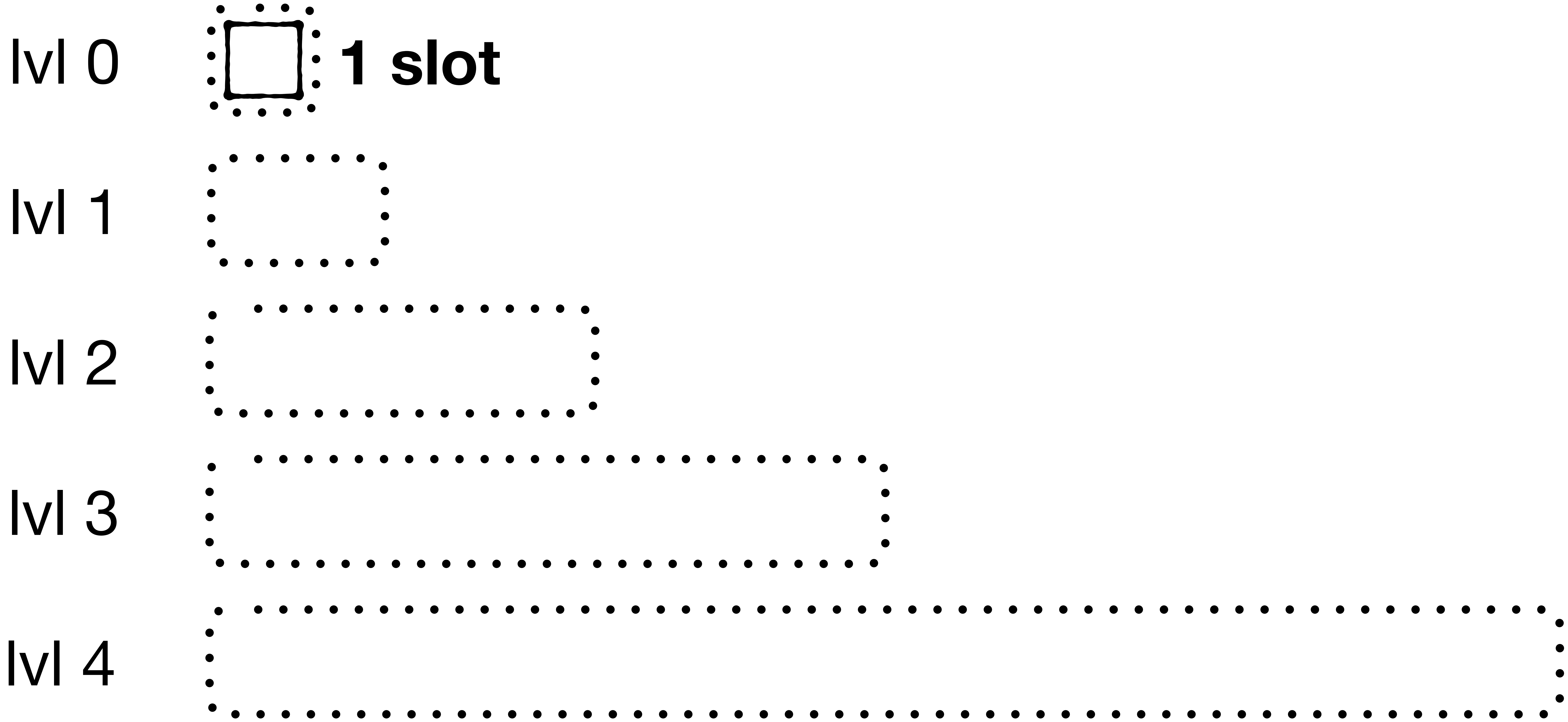


Challenges: How to grow blocks to meet these properties?

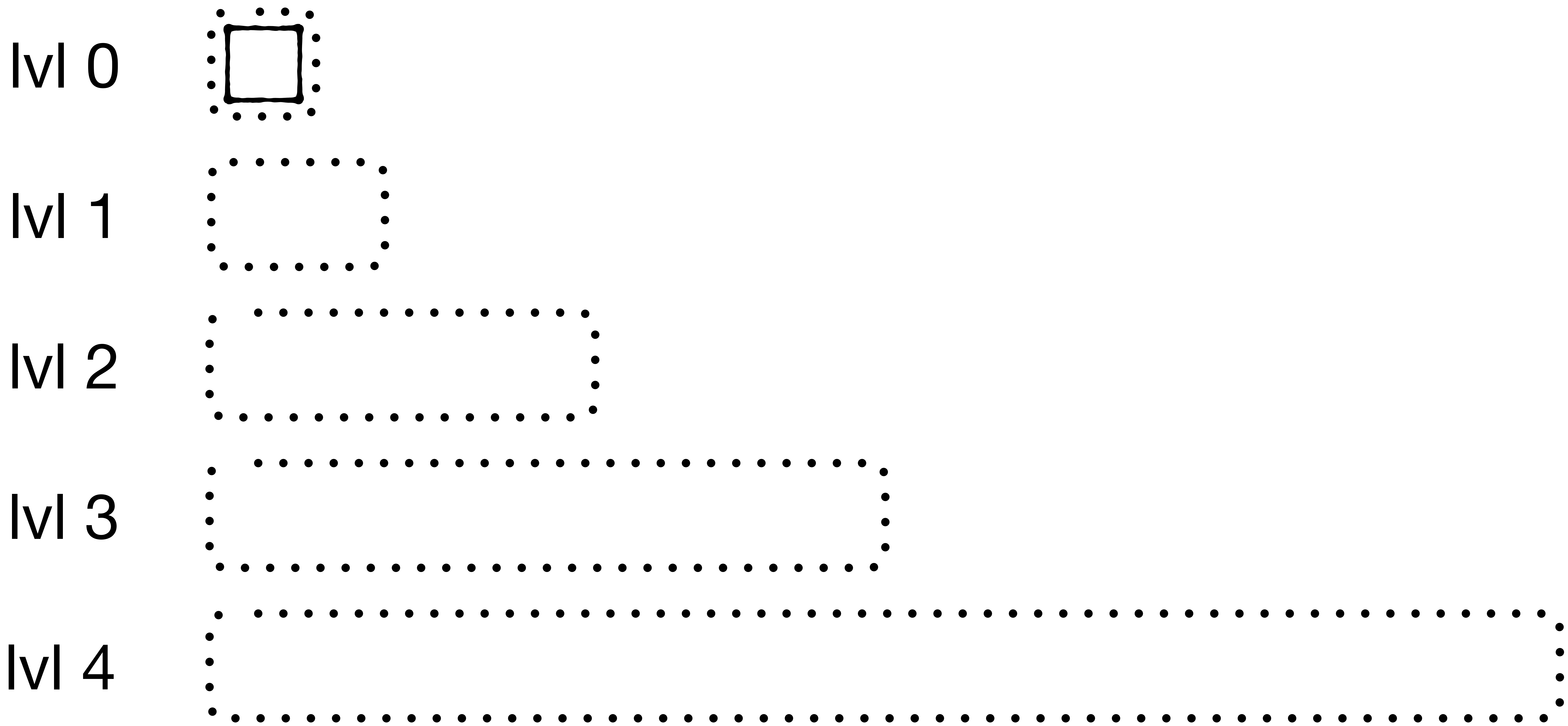
Inferring which block contains which array offset?

Multiple levels



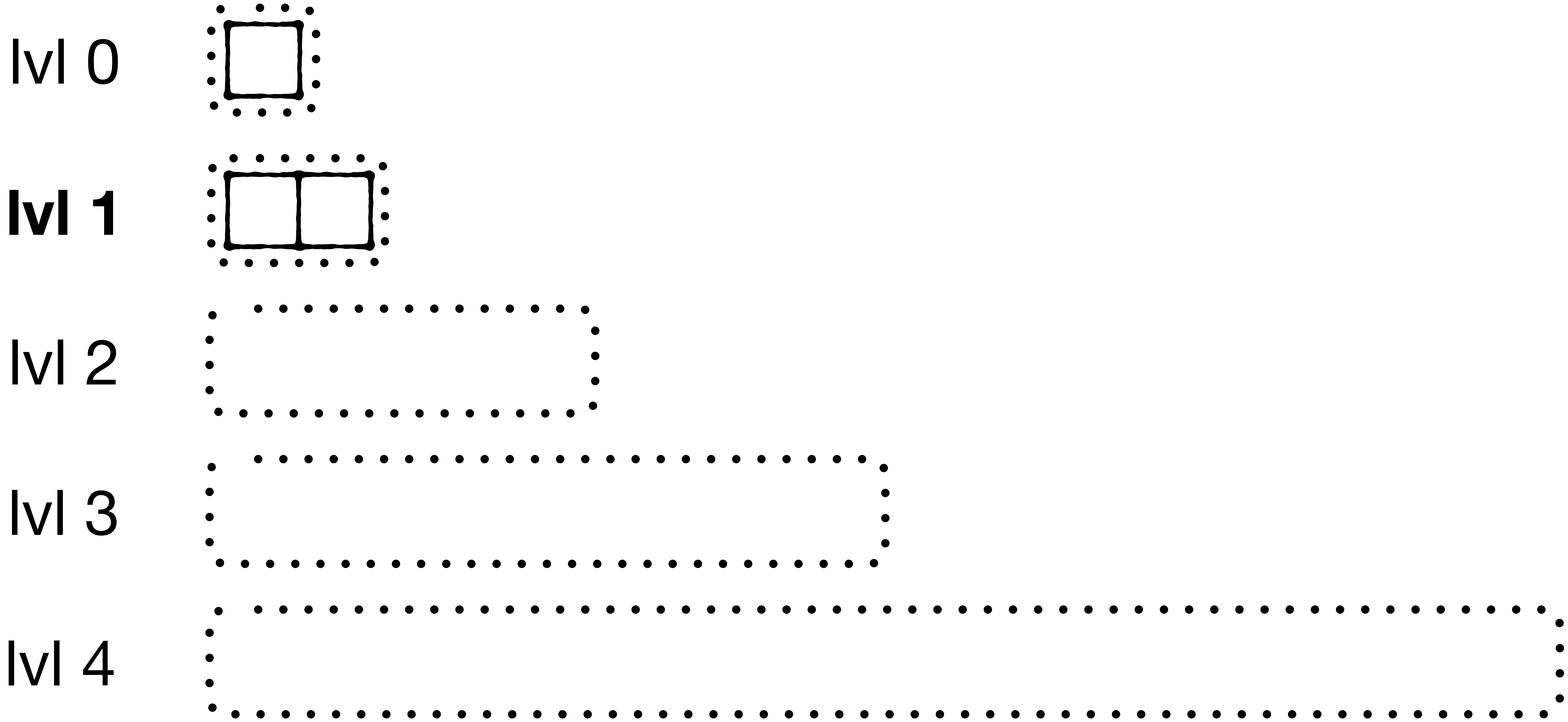

lvl 0
1 slot
lvl 1
lvl 2
lvl 3
lvl 4

In every pair of subsequent levels k and k+1



In every pair of subsequent levels k and k+1

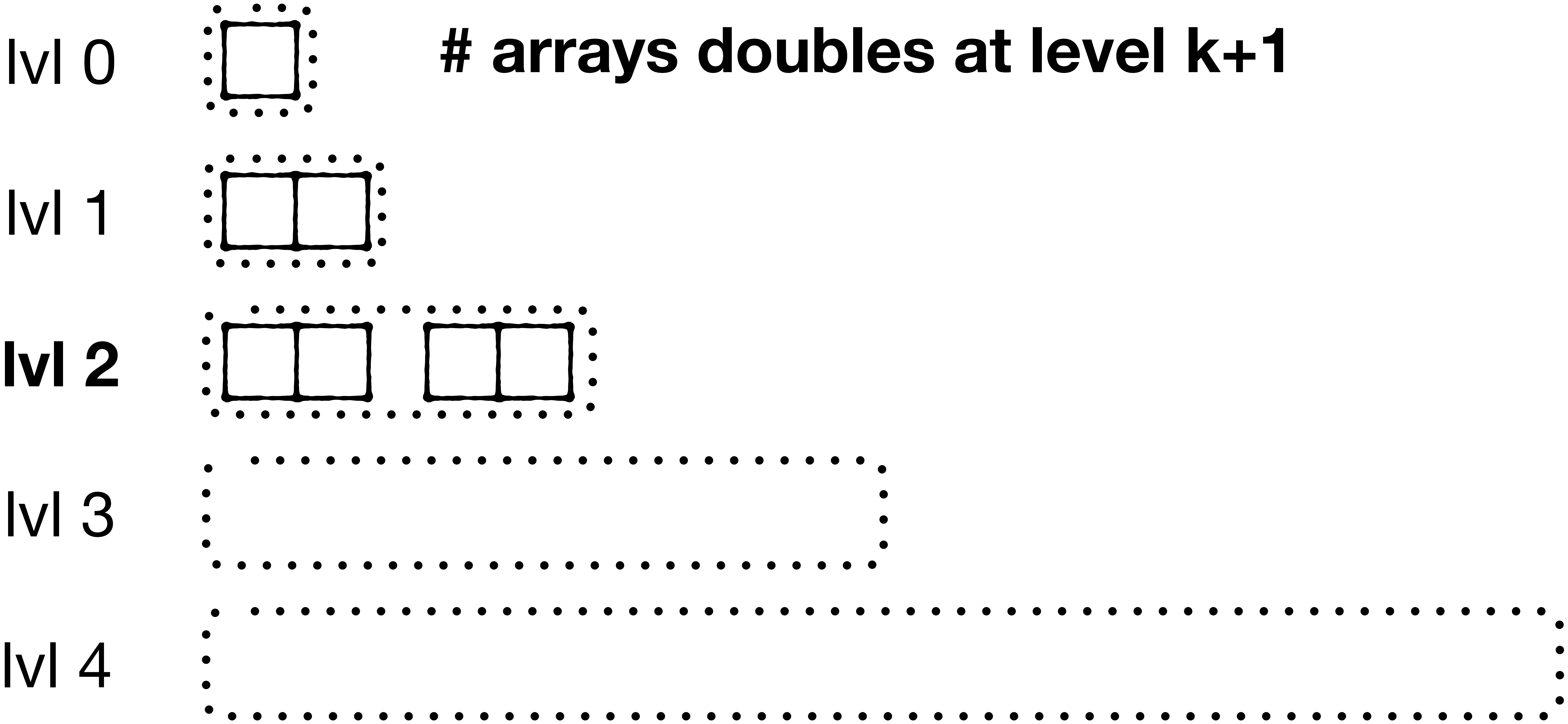
Size of arrays doubles at level k



In every pair of subsequent levels k and k+1

Size of arrays doubles at level k

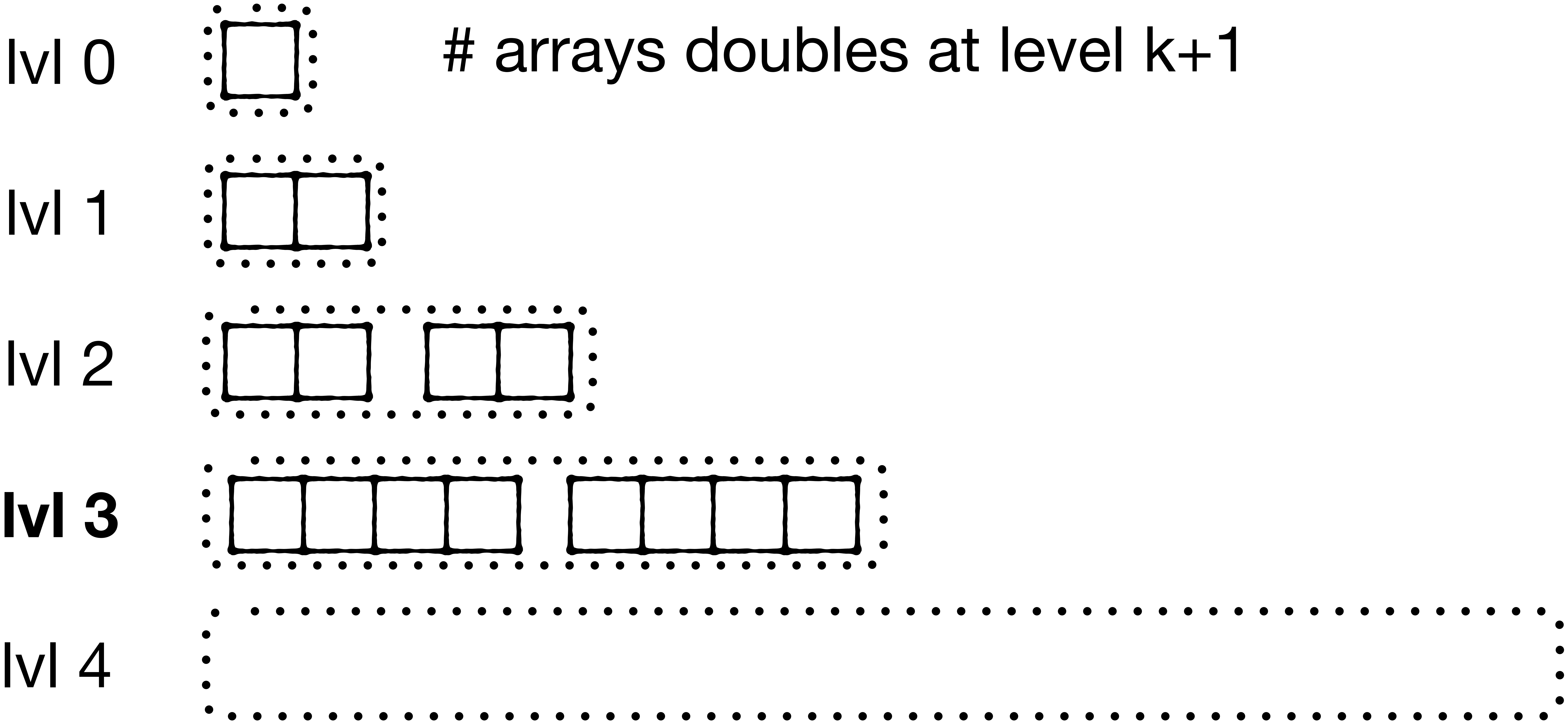
 # arrays doubles at level k+1



In every pair of subsequent levels k and k+1

Size of arrays doubles at level k

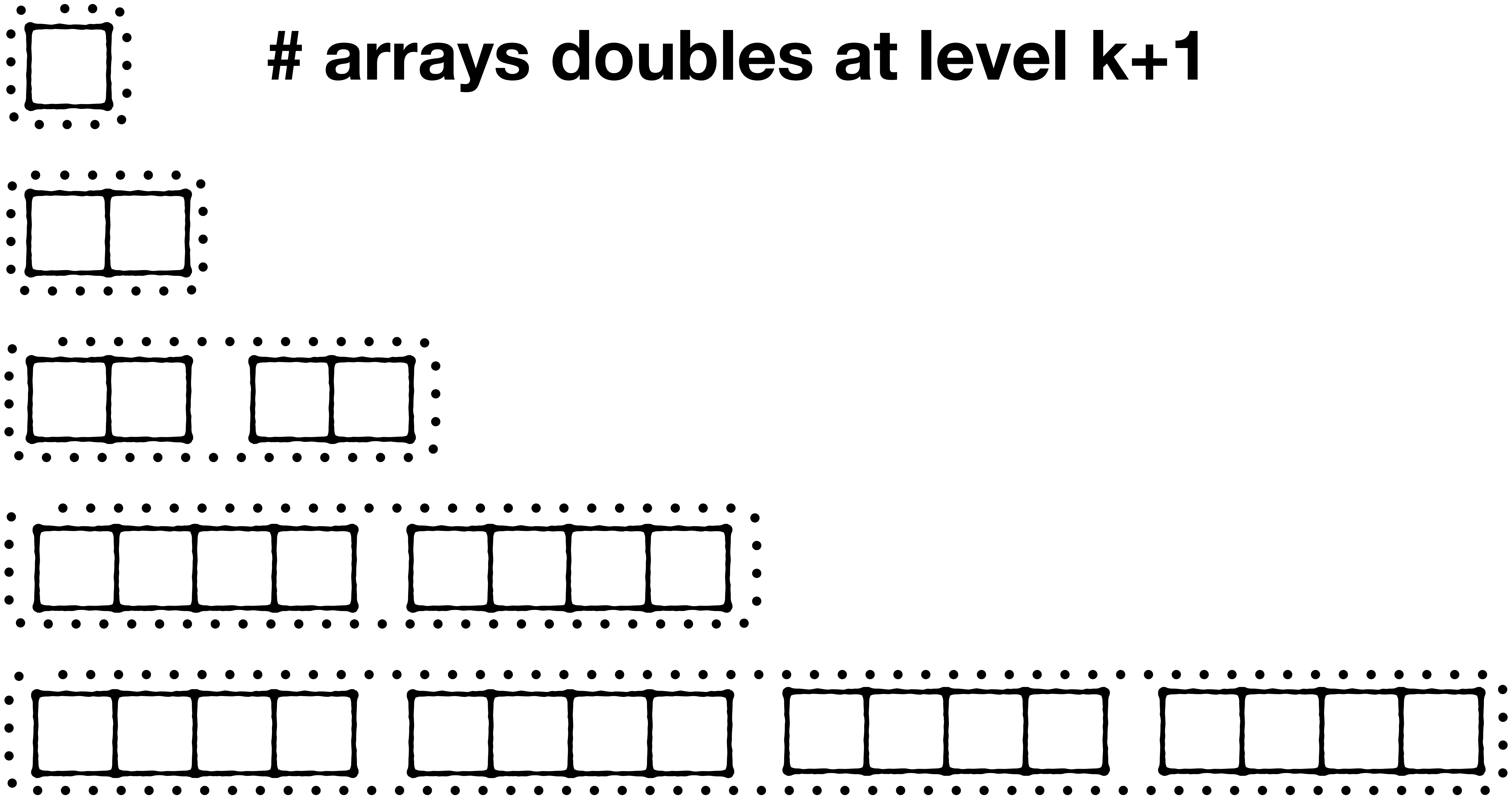
arrays doubles at level k+1



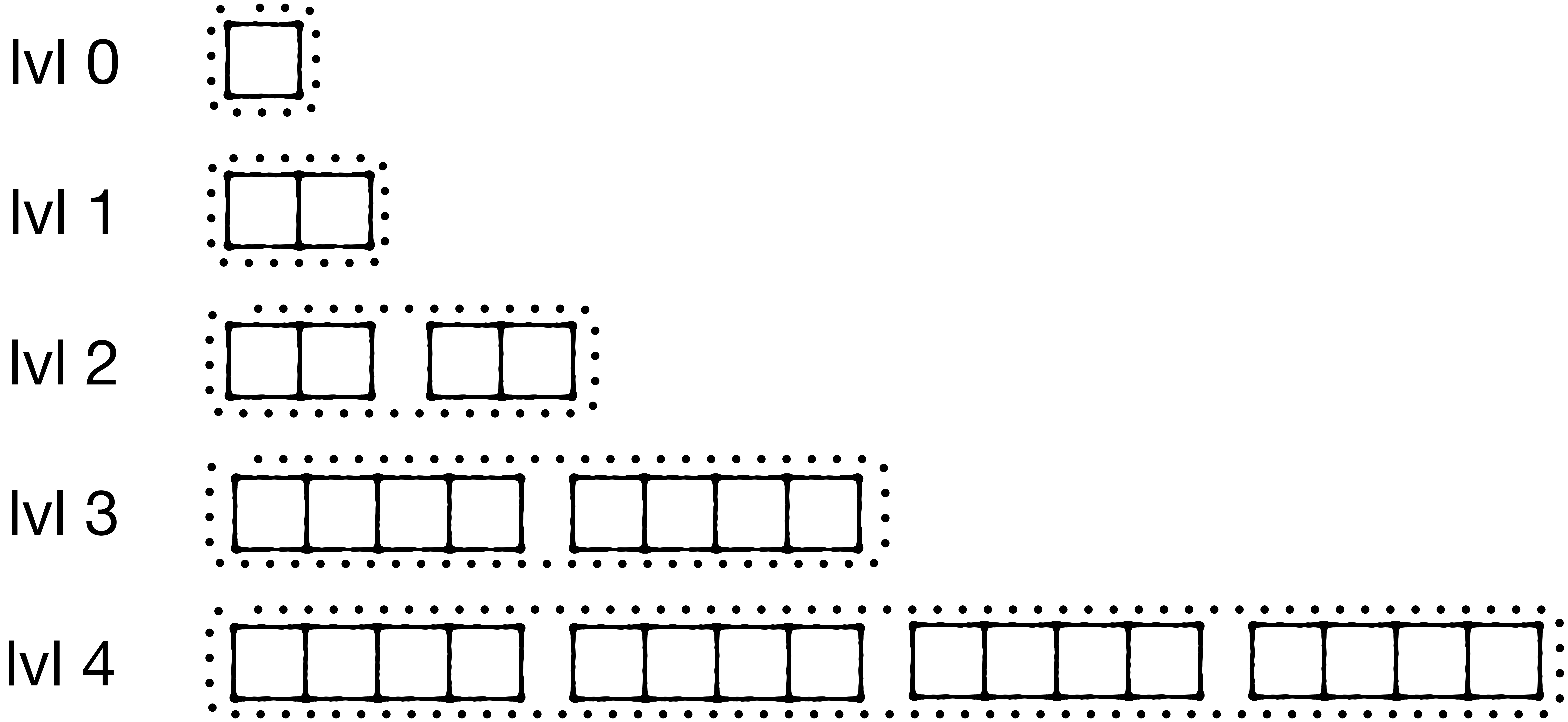
In every pair of subsequent levels k and k+1

Size of arrays doubles at level k

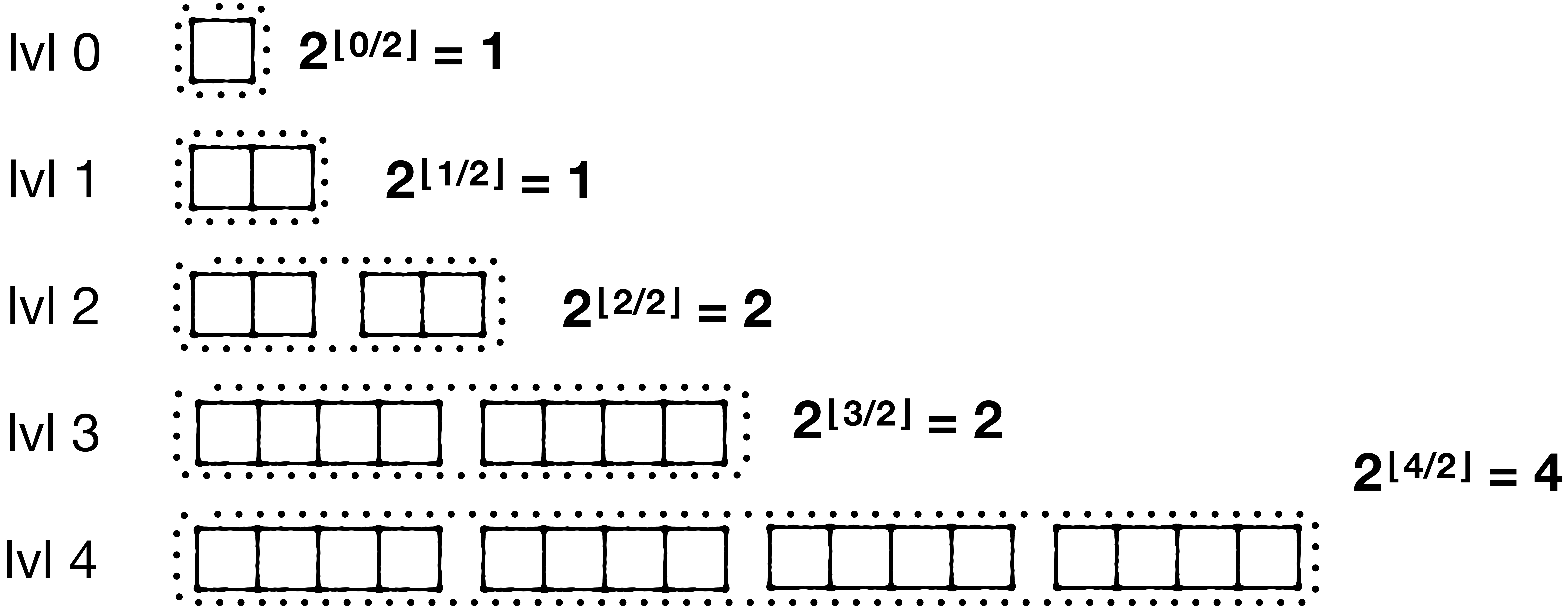
 # arrays doubles at level k+1



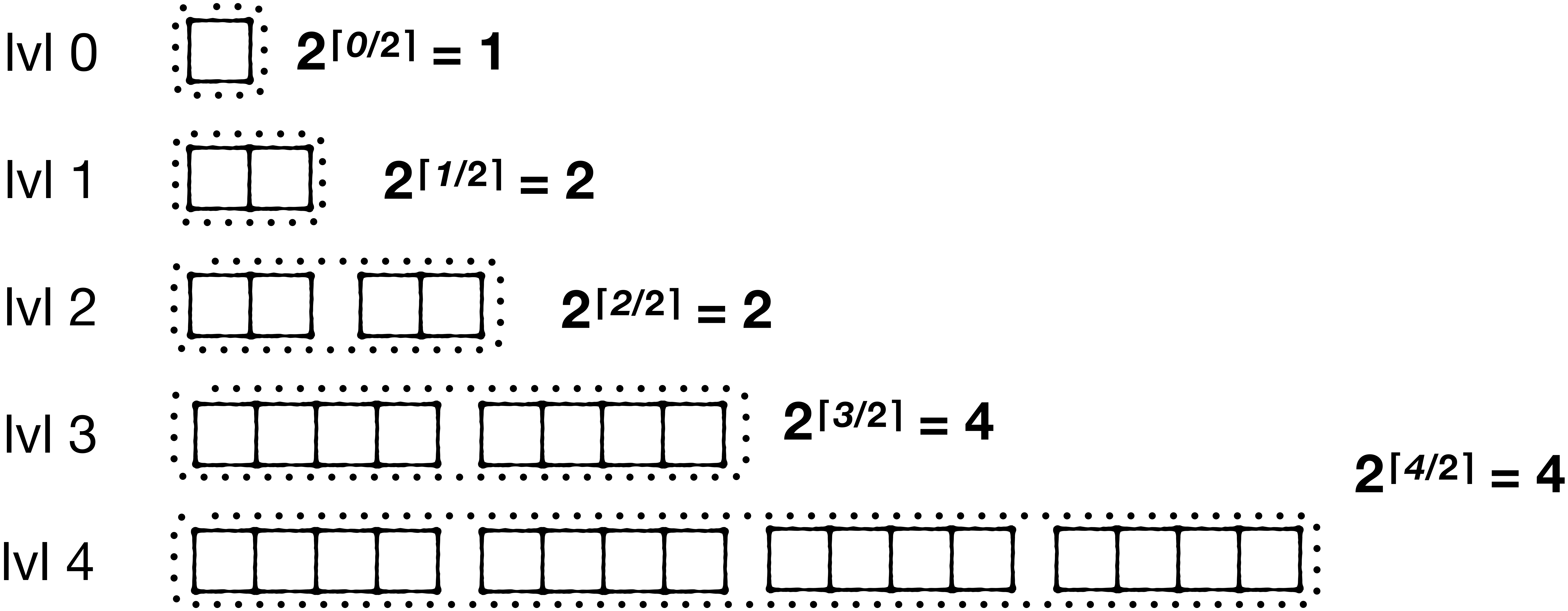
lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

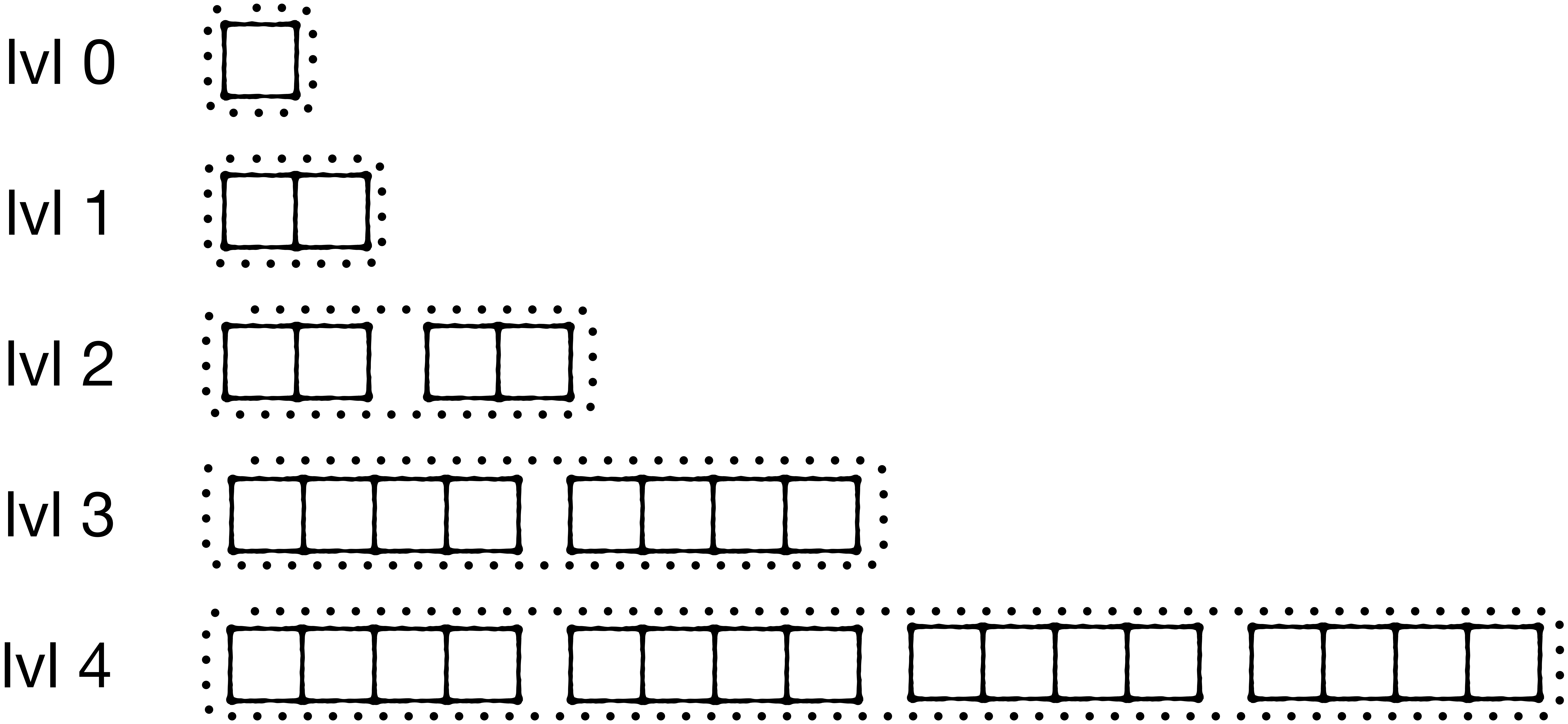


lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, **each with $2^{\lceil K/2 \rceil}$ slots**



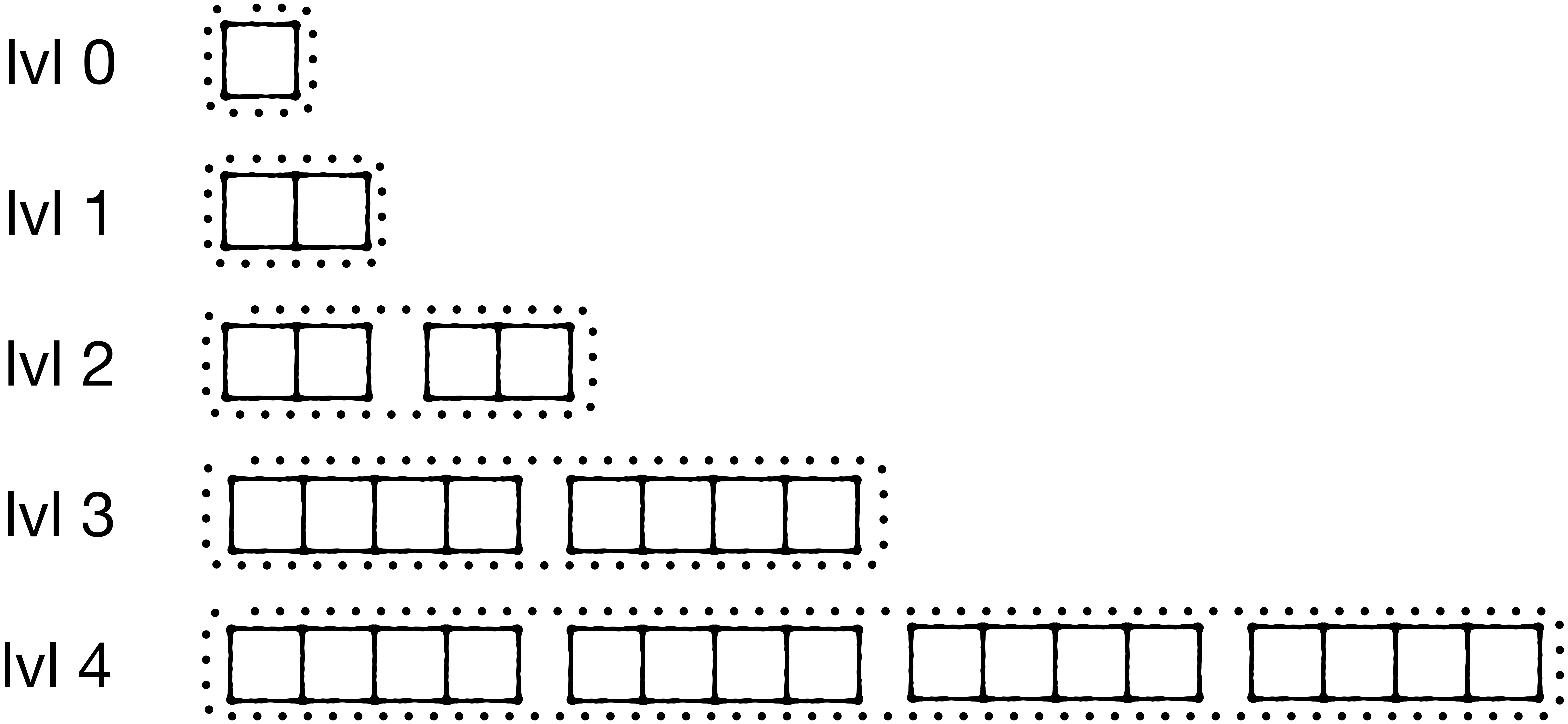
lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

 # levels?



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

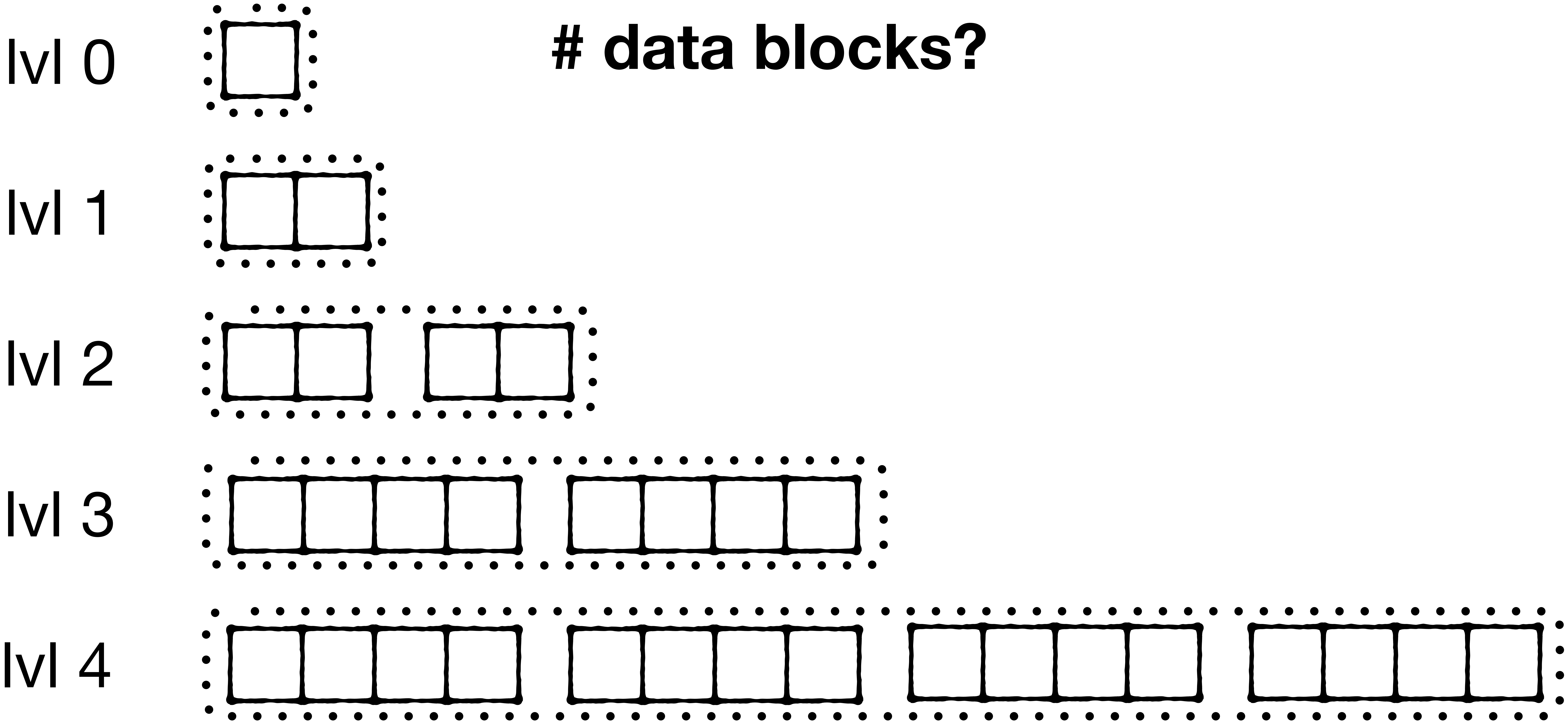
 # levels: $\log_2 N$



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

\# levels: $\log_2 N$

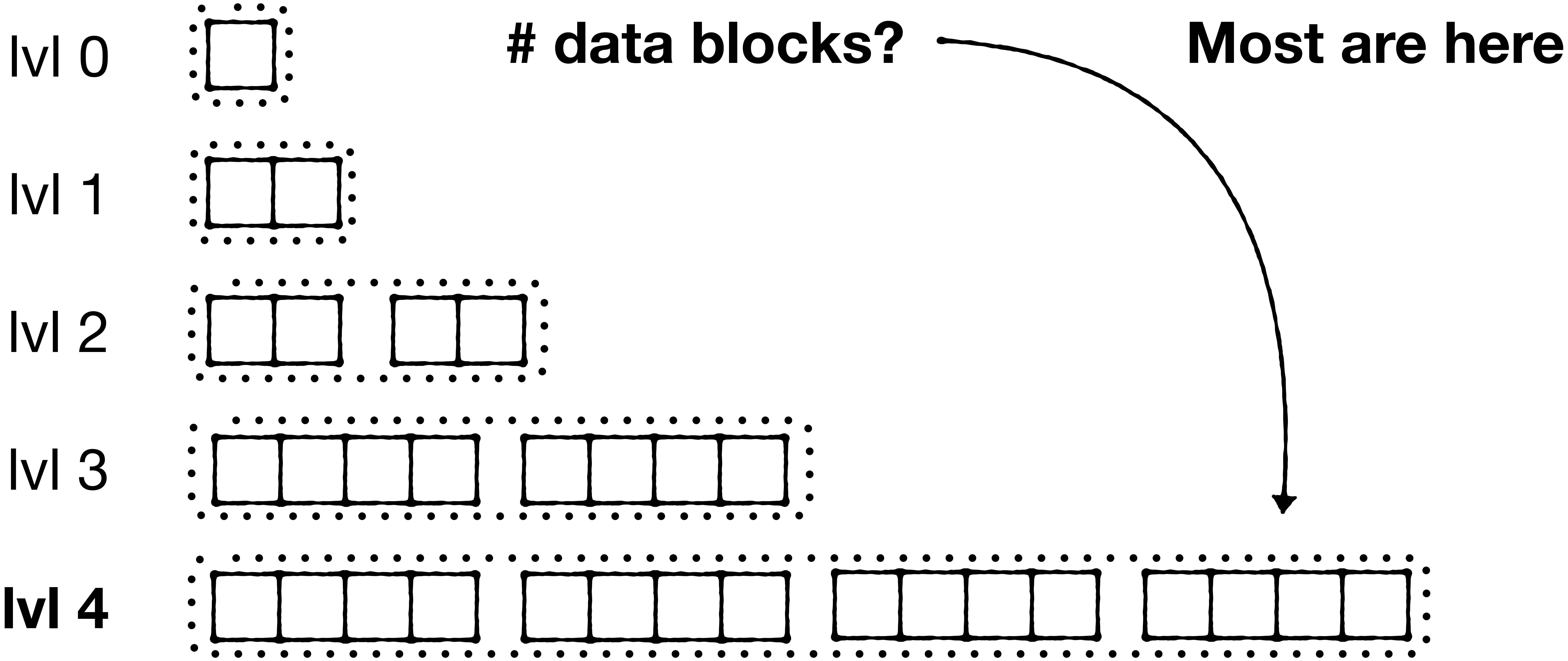
 # data blocks?



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

levels: $\log_2 N$

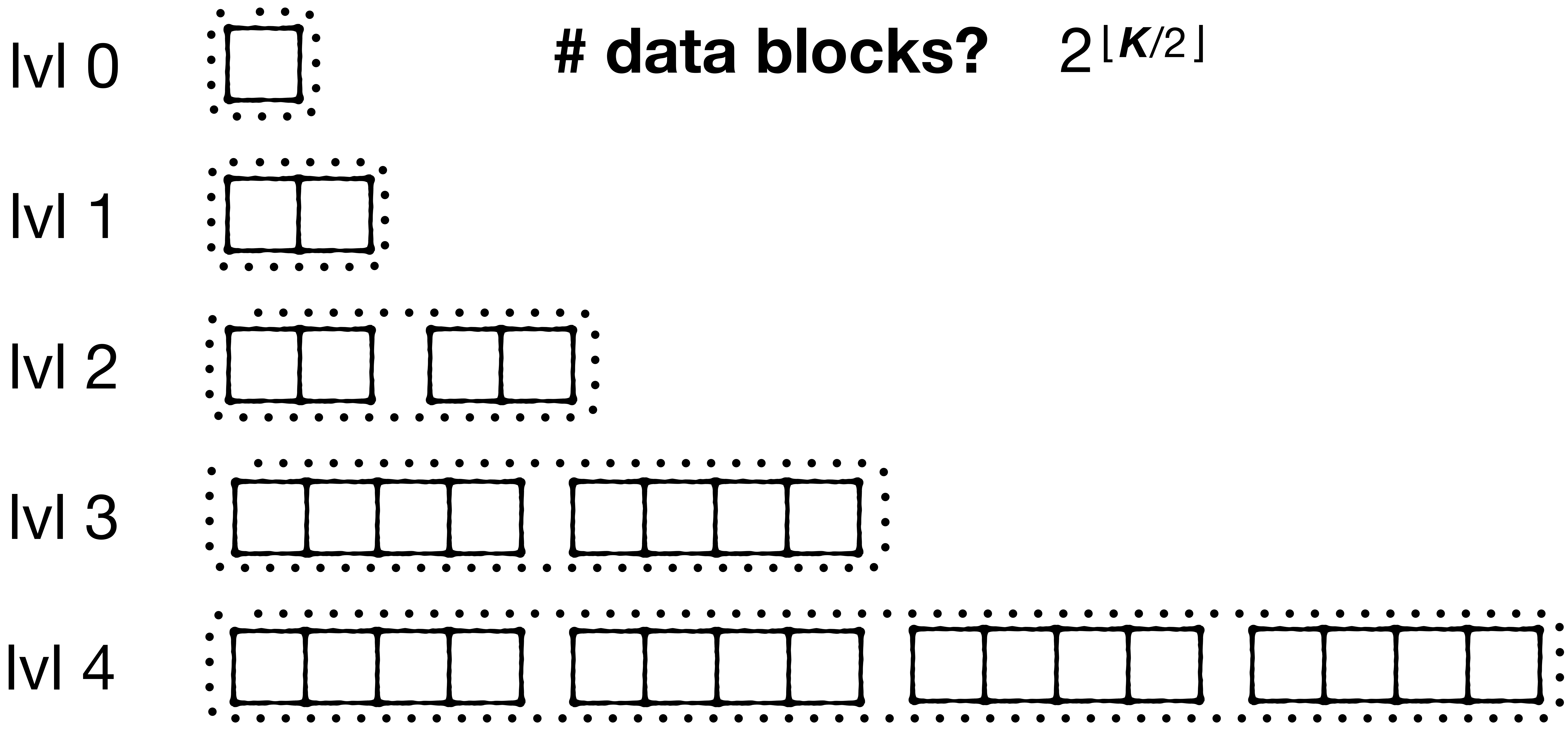
 # data blocks?



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

levels: **$\log_2 N$**

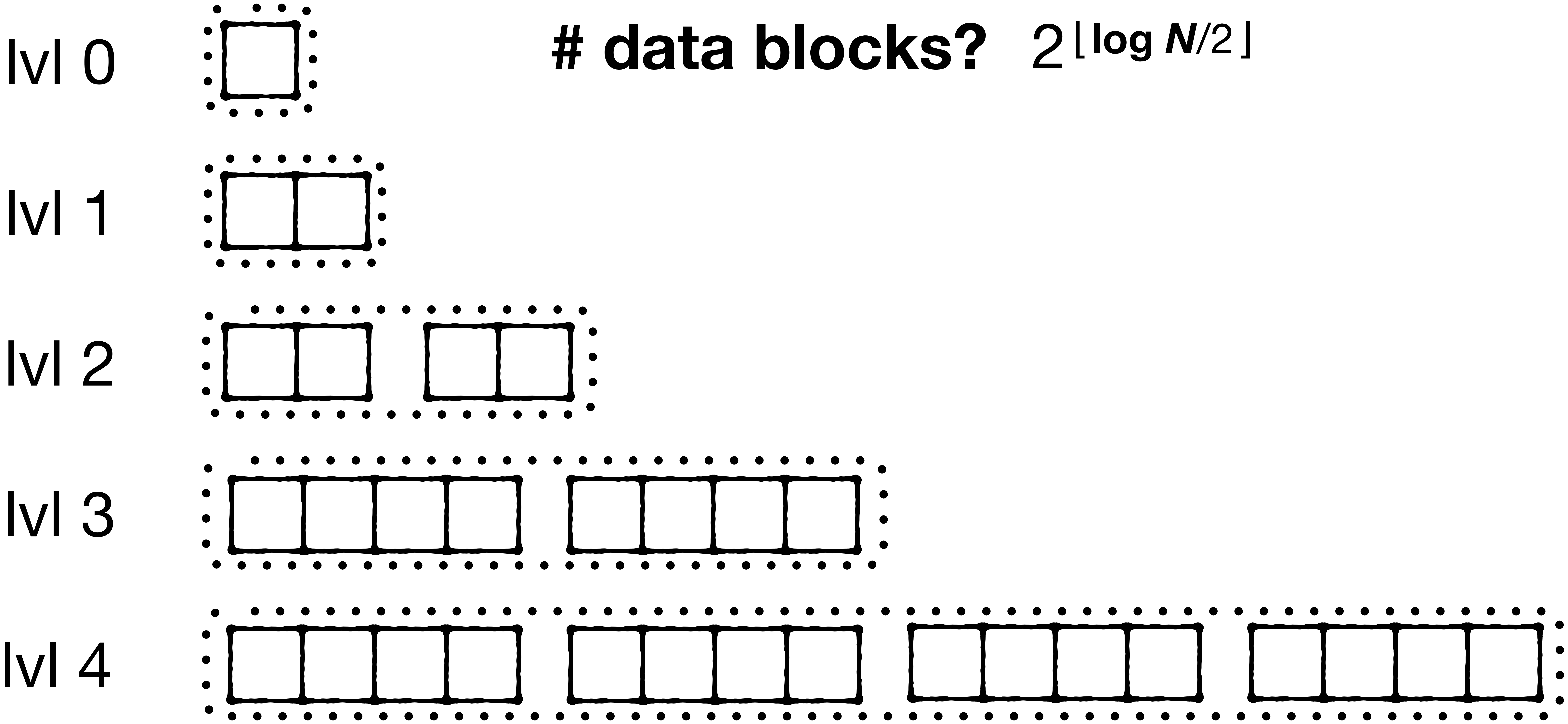
 # data blocks? $2^{\lfloor \boldsymbol{K}/2 \rfloor}$



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

levels: **$\log_2 N$**

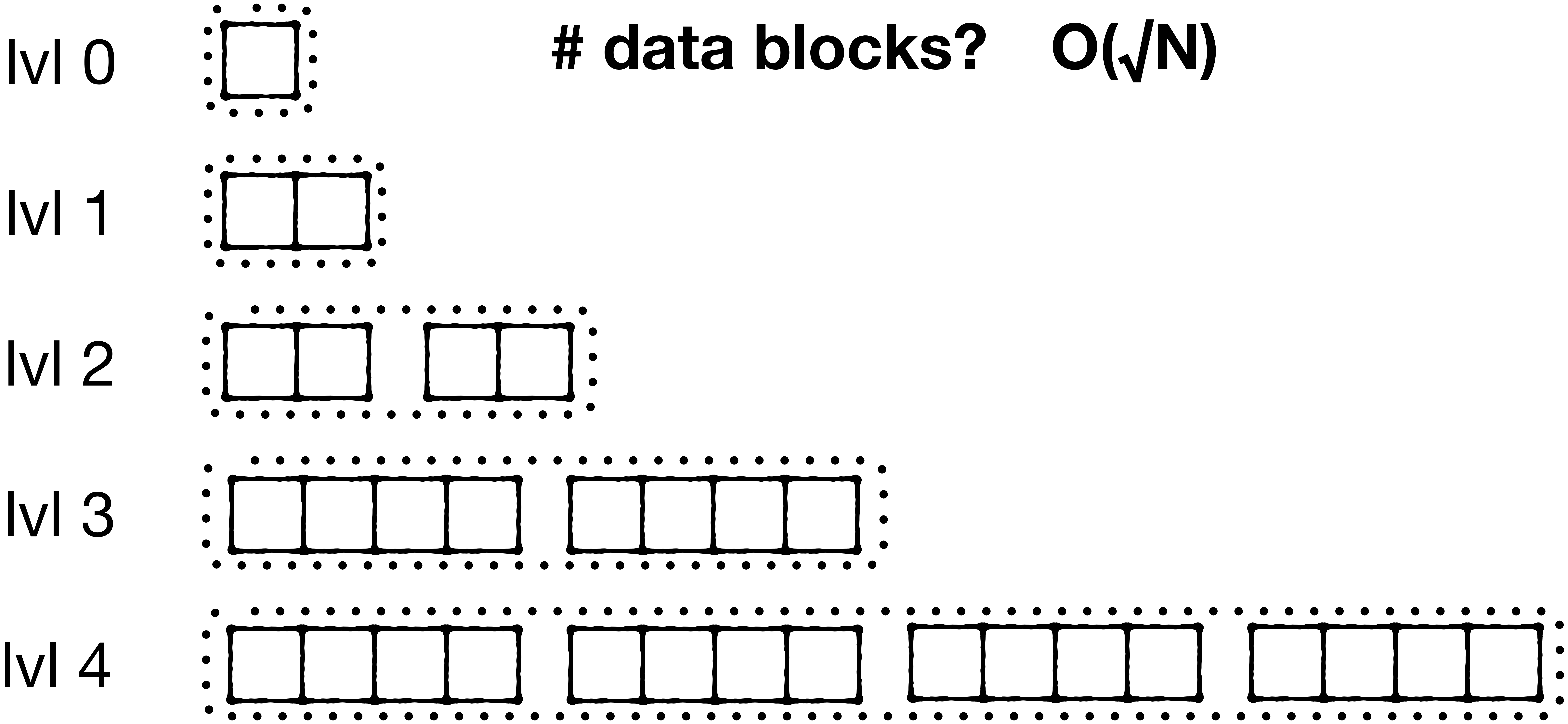
 # data blocks? $2^{\lfloor \log N/2 \rfloor}$



lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

levels: $\log_2 N$

 # data blocks? O(√N)

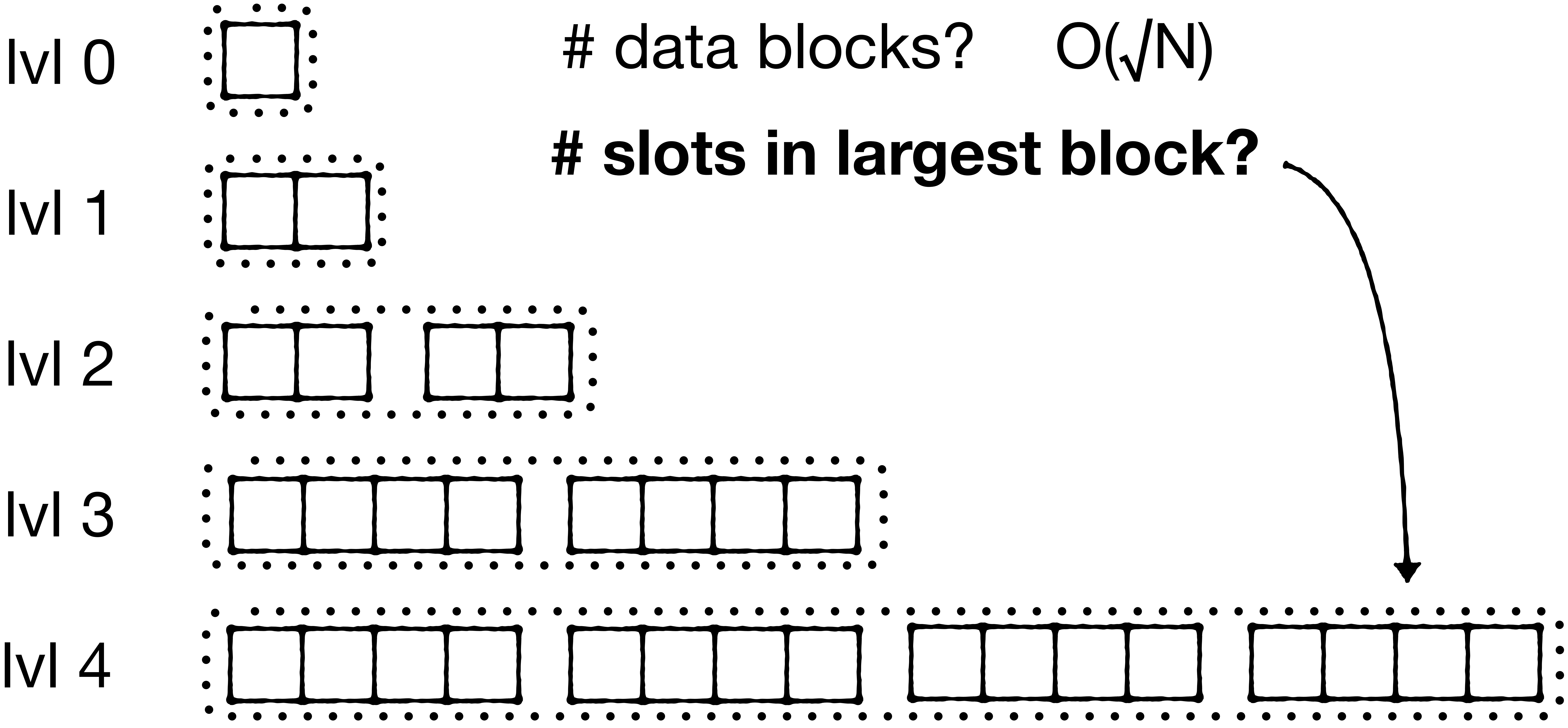


lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

levels: $\log_2 N$

data blocks? $O(\sqrt{N})$

 # slots in largest block?

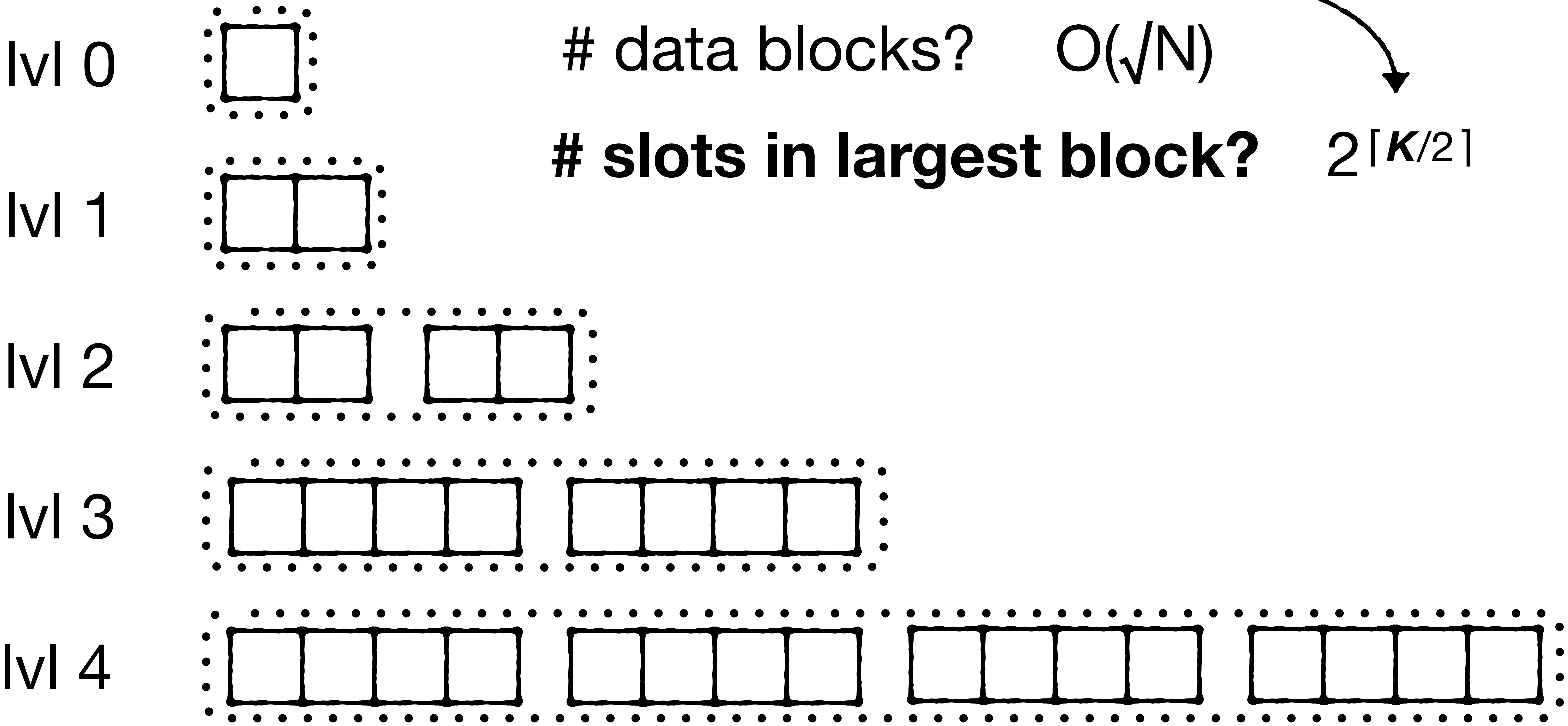


lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

\# levels: **$\log_2 N$**

\# data blocks? $O(\sqrt{N})$

 # slots in largest block? $2^{\lceil \boldsymbol{K}/2 \rceil}$

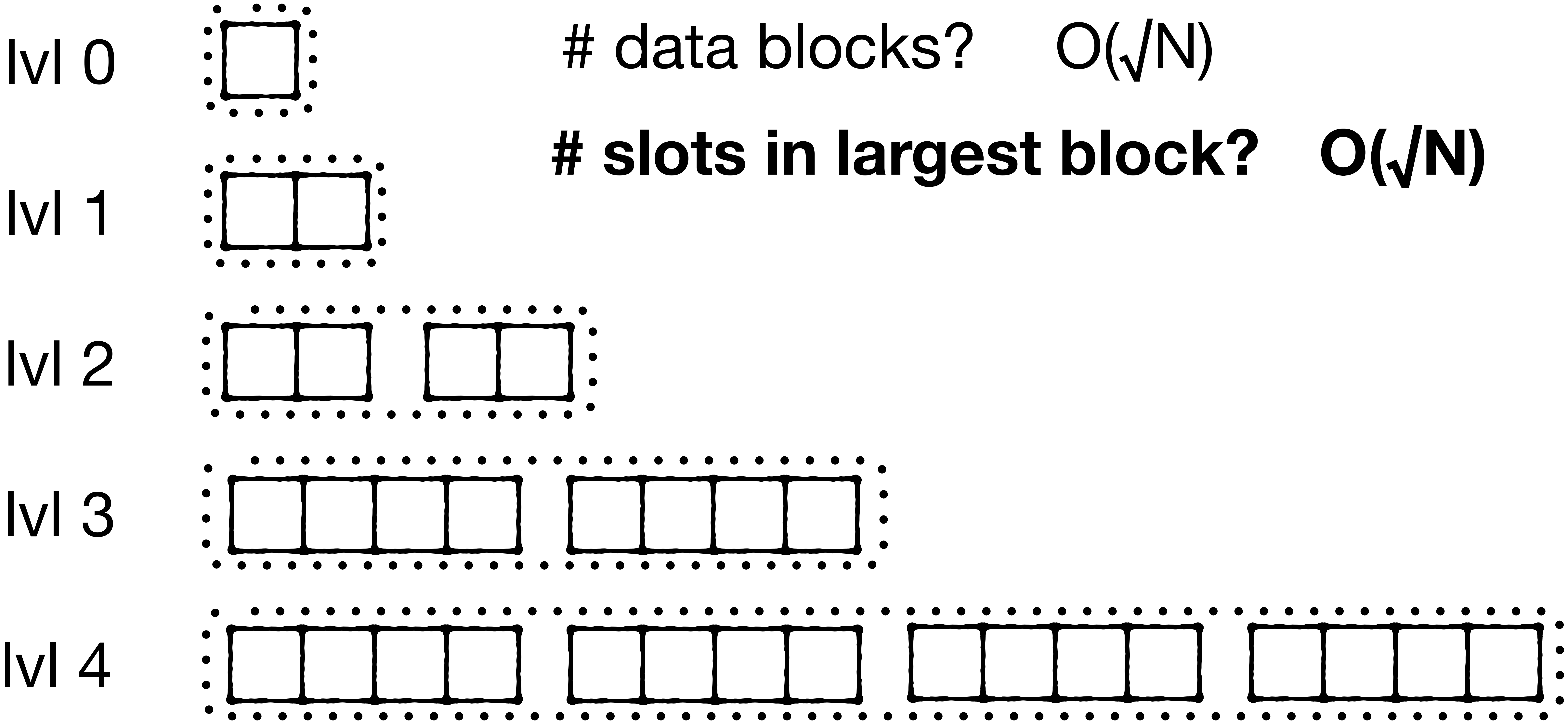


lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

\# levels: $\log_2 N$

\# data blocks? $O(\sqrt{N})$

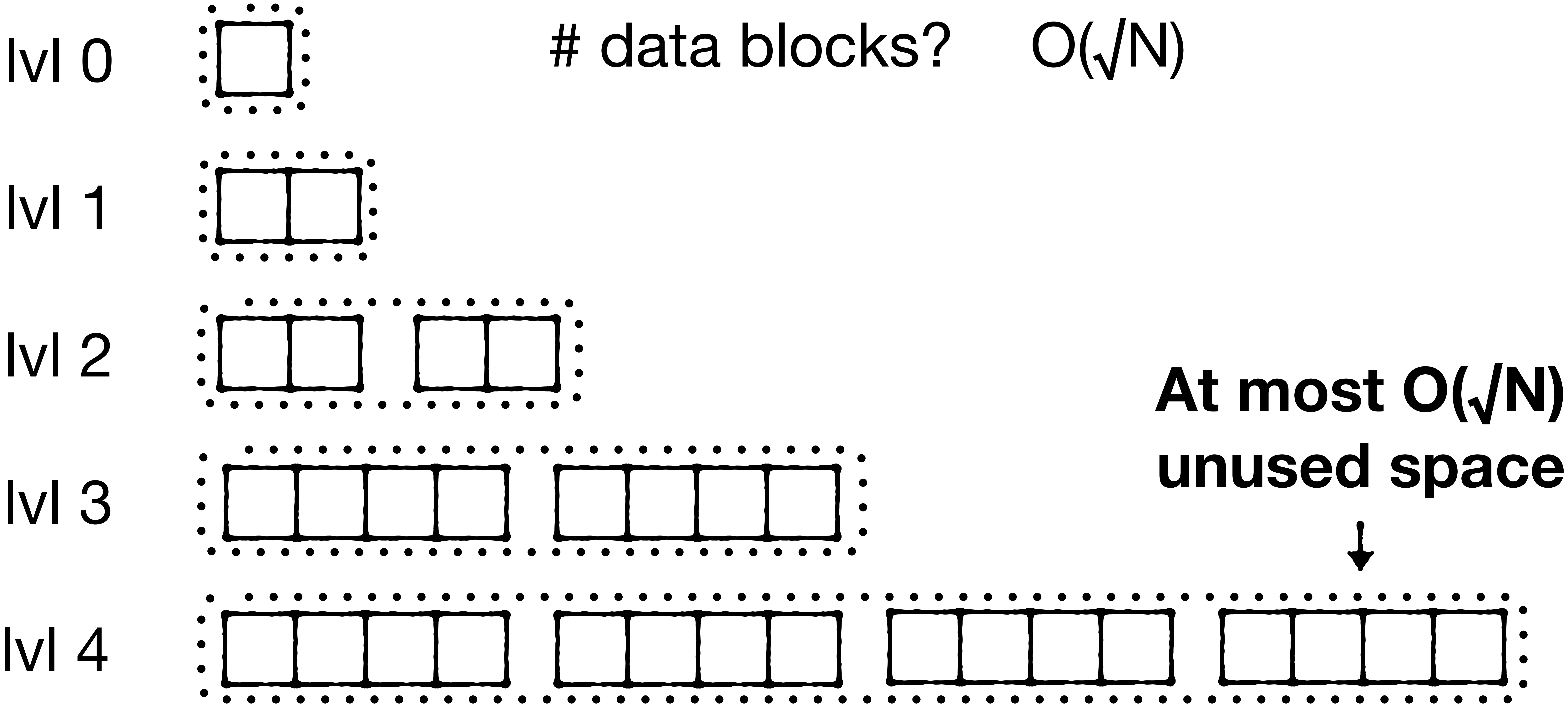
\# slots in largest block? $O(\sqrt{N})$

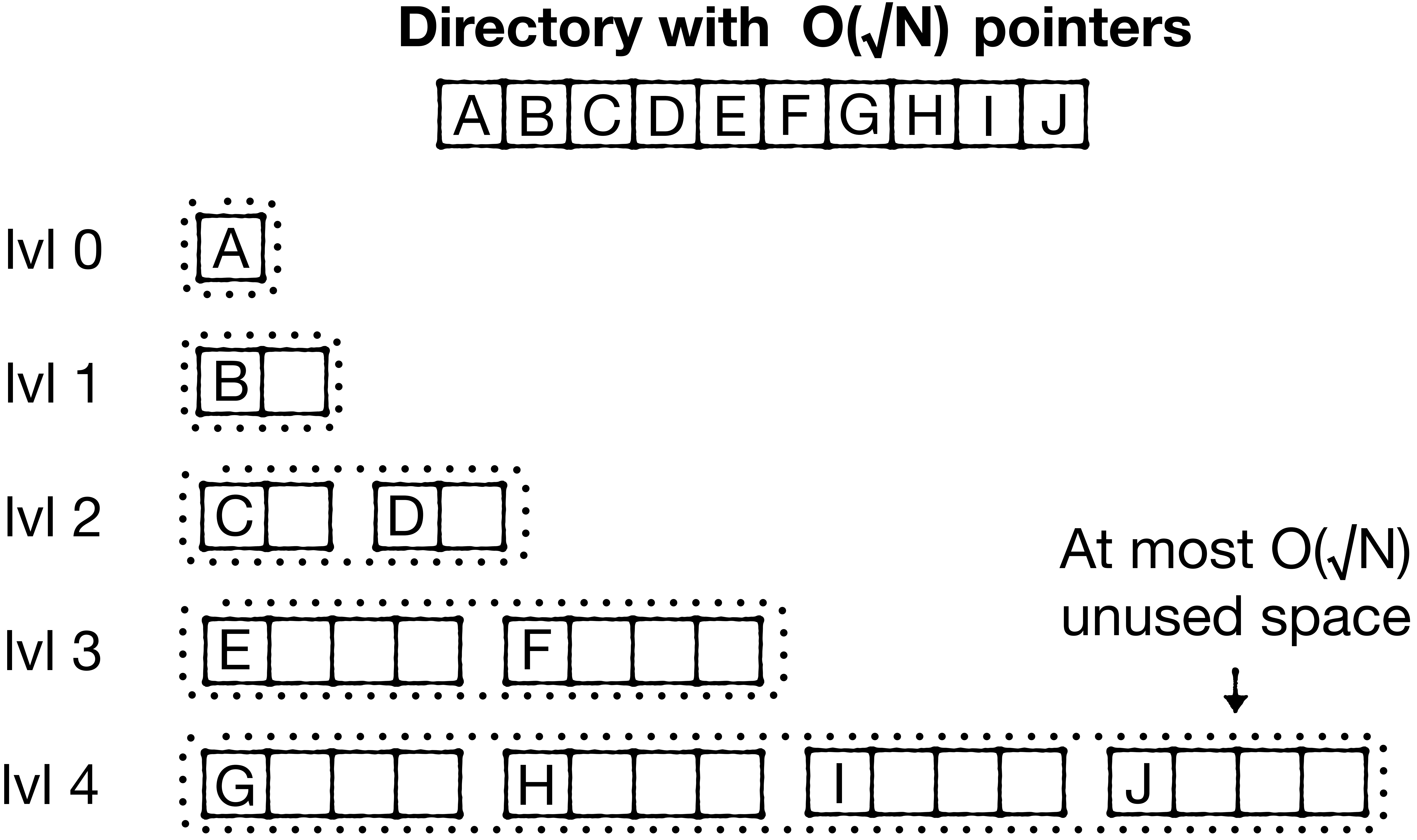


lvl i contains $2^{\lfloor K/2 \rfloor}$ blocks, each with $2^{\lceil K/2 \rceil}$ slots

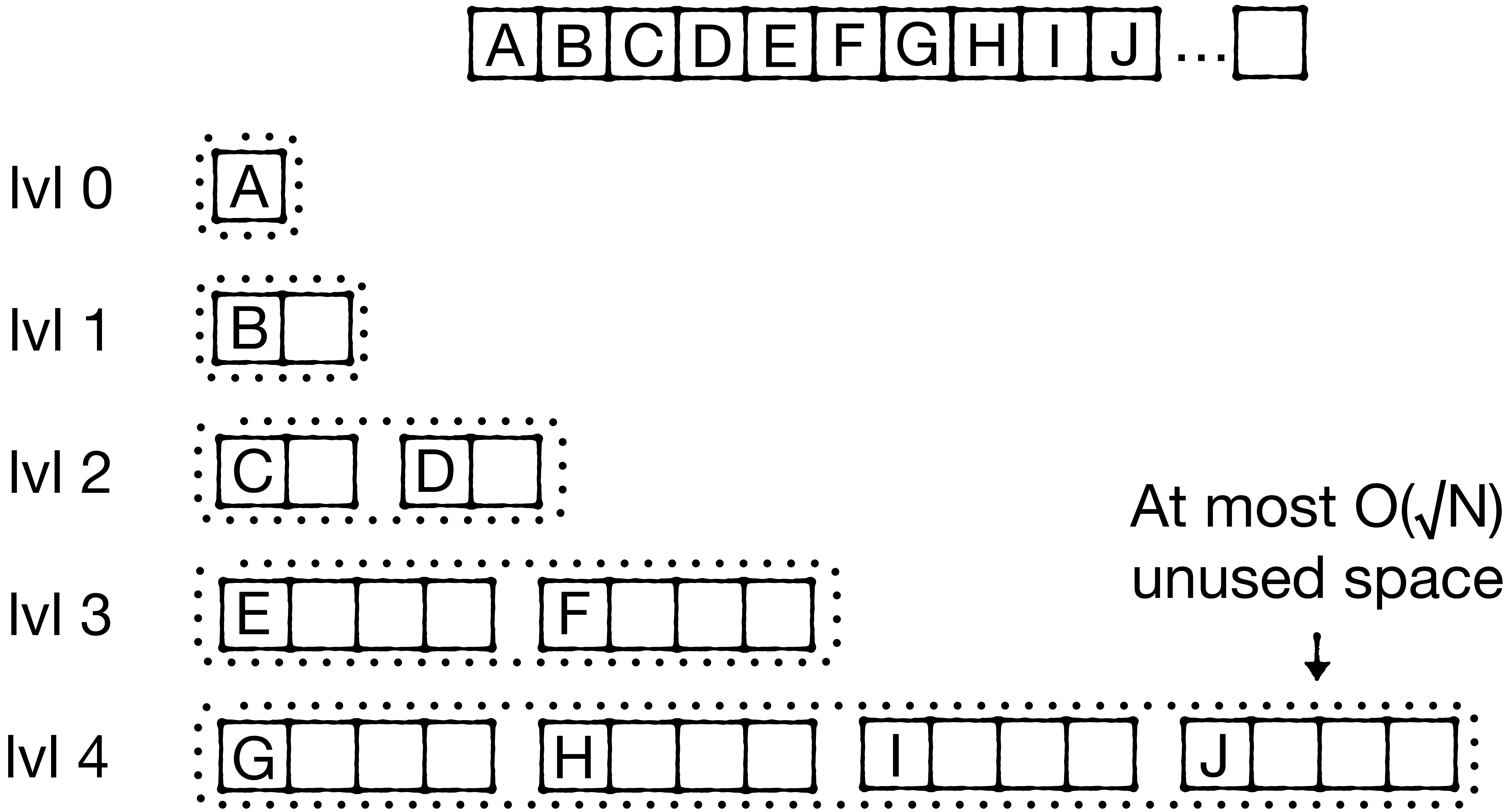
\# levels: $\log_2 N$

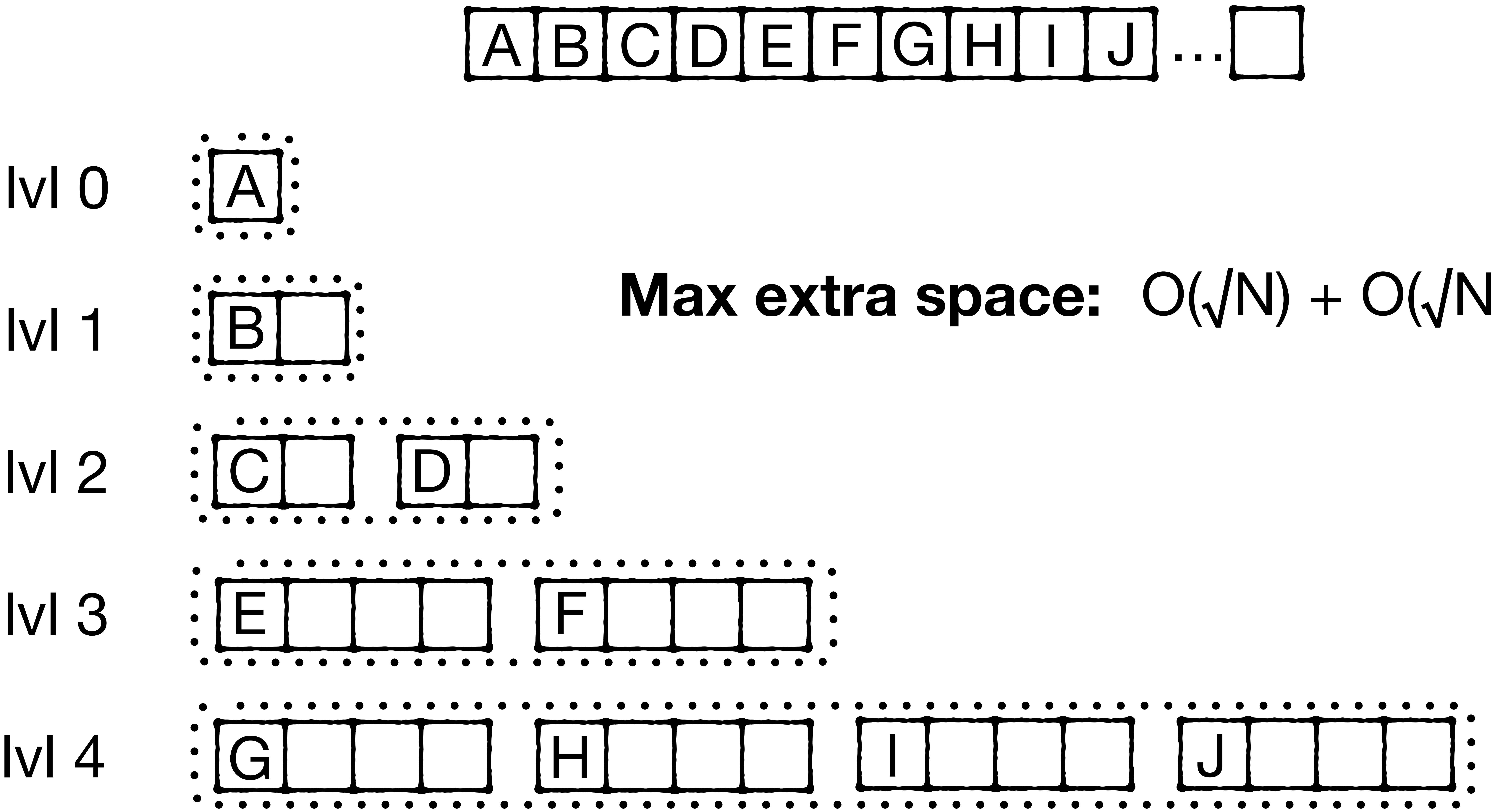
\# data blocks? $O(\sqrt{N})$



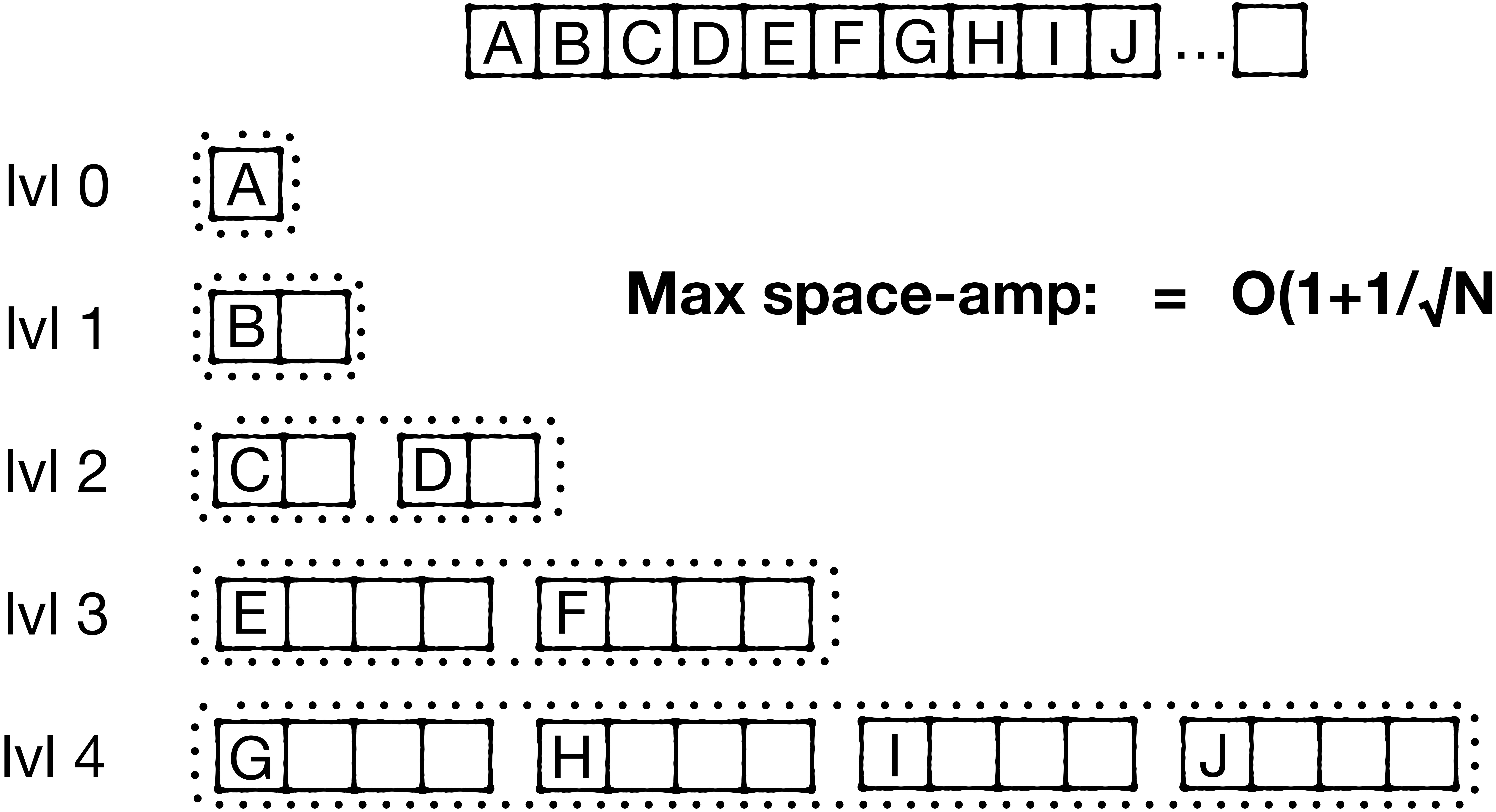

Directory with O(√N) pointers
A B C D E F G H I J
lvl 0
A
lvl 1
B
lvl 2
C
D
lvl 3
E
F
lvl 4
G
H
I
J
At most O(√N) unused space

At most half O(√N) unused space



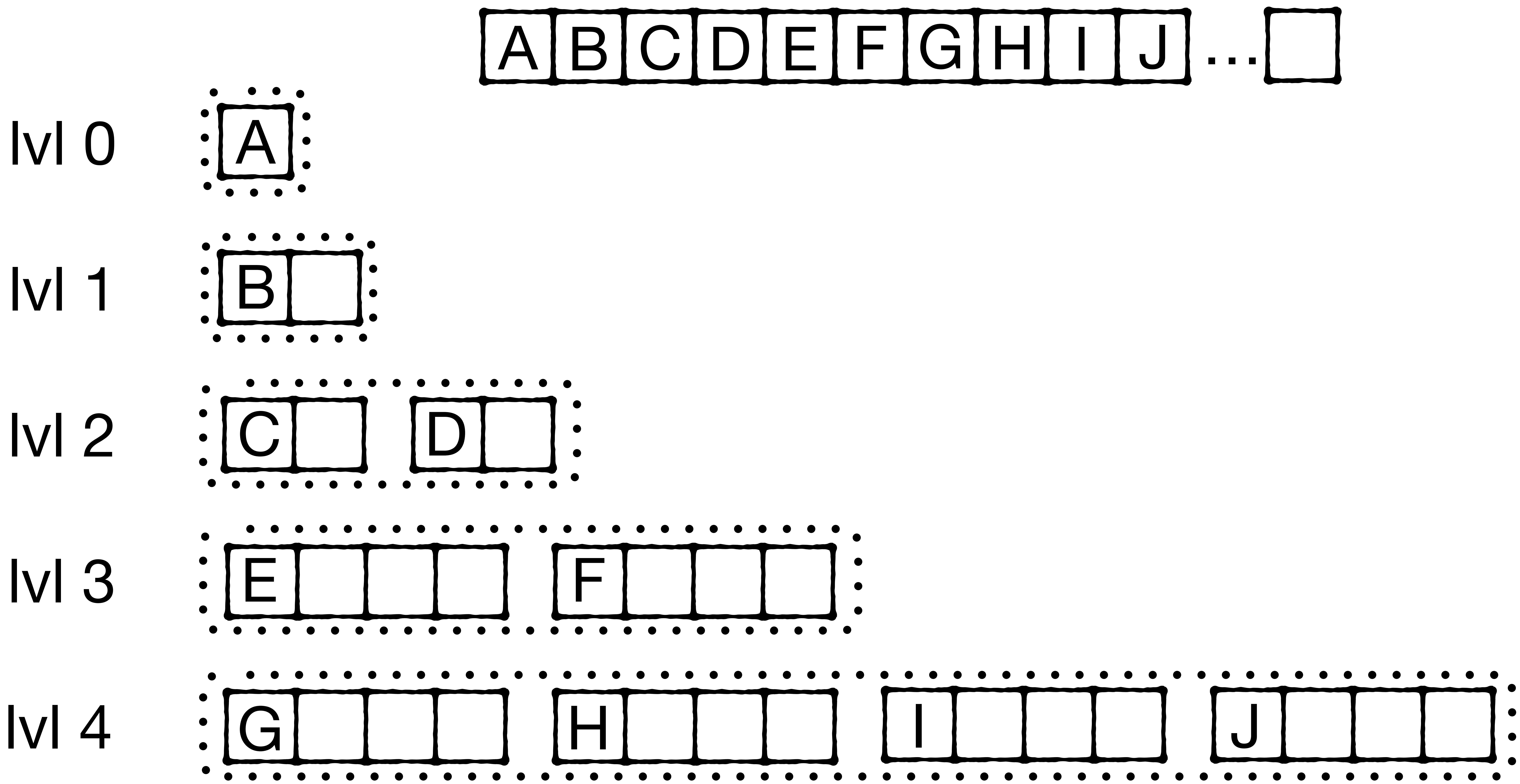


Max extra space: $O(\sqrt{N}) + O(\sqrt{N})$ **= $O(\sqrt{N})$**

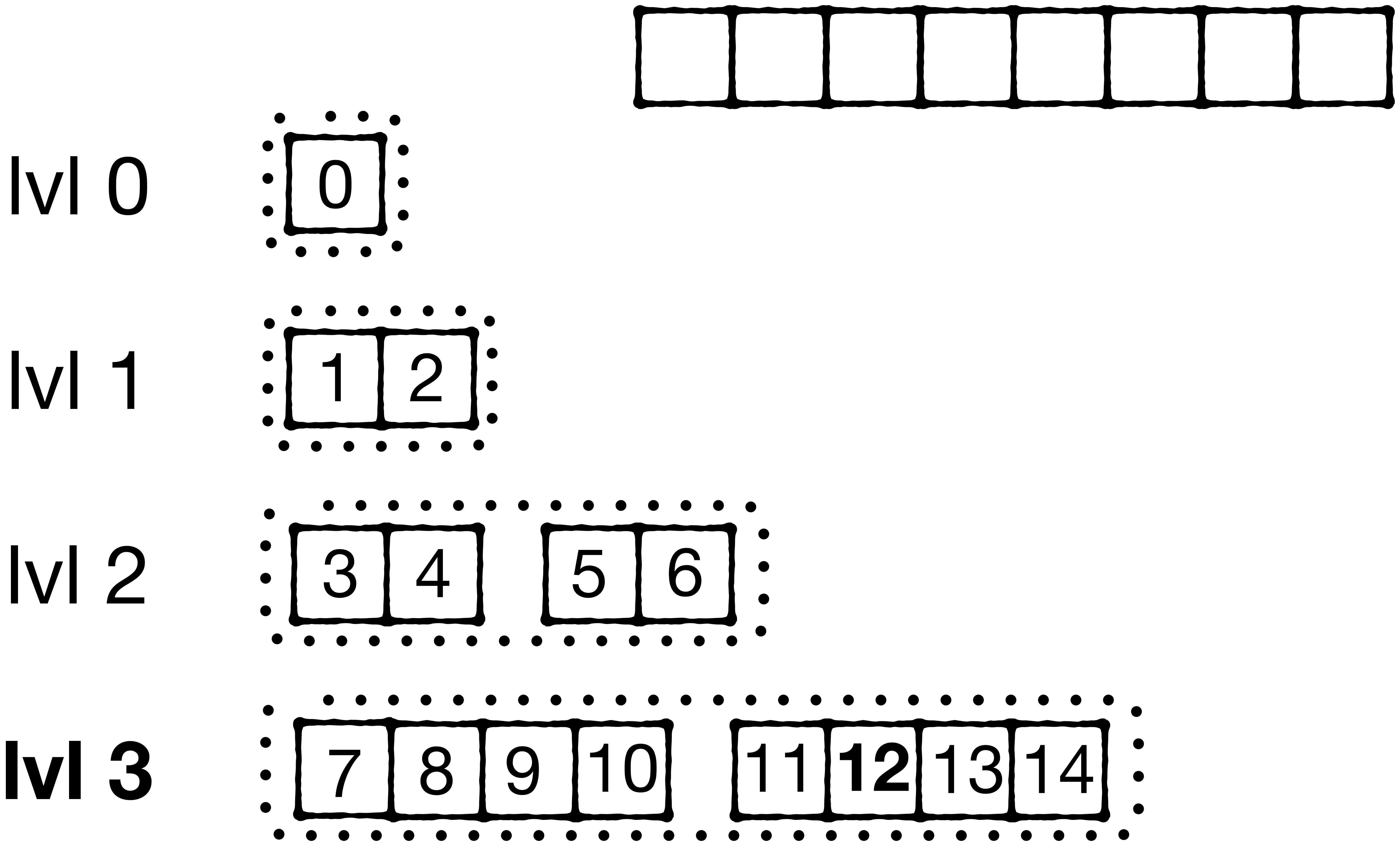


Max space-amp: = $O(1+1/\sqrt{N})$

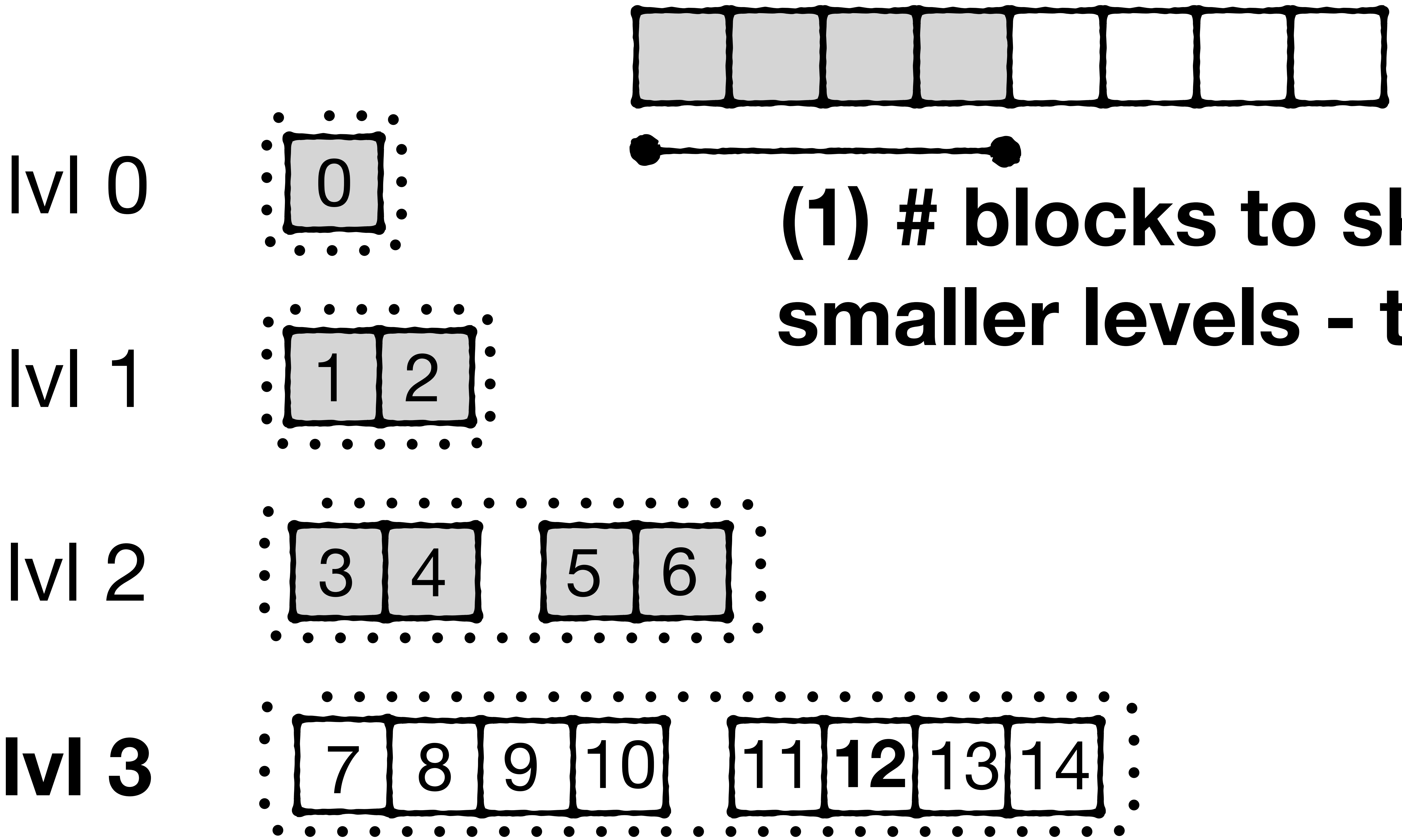
How to access slot in O(1) time?



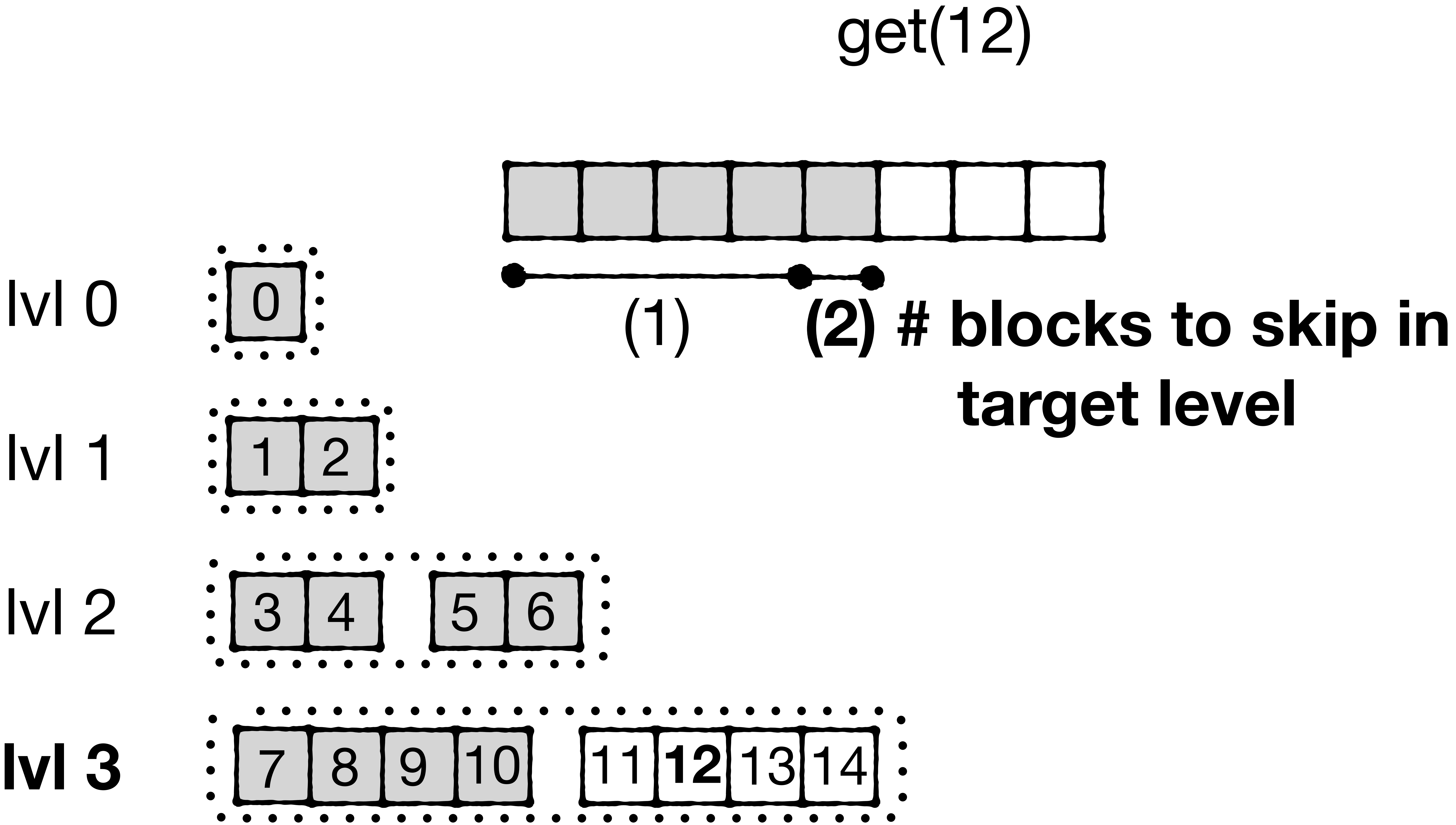
get(12)

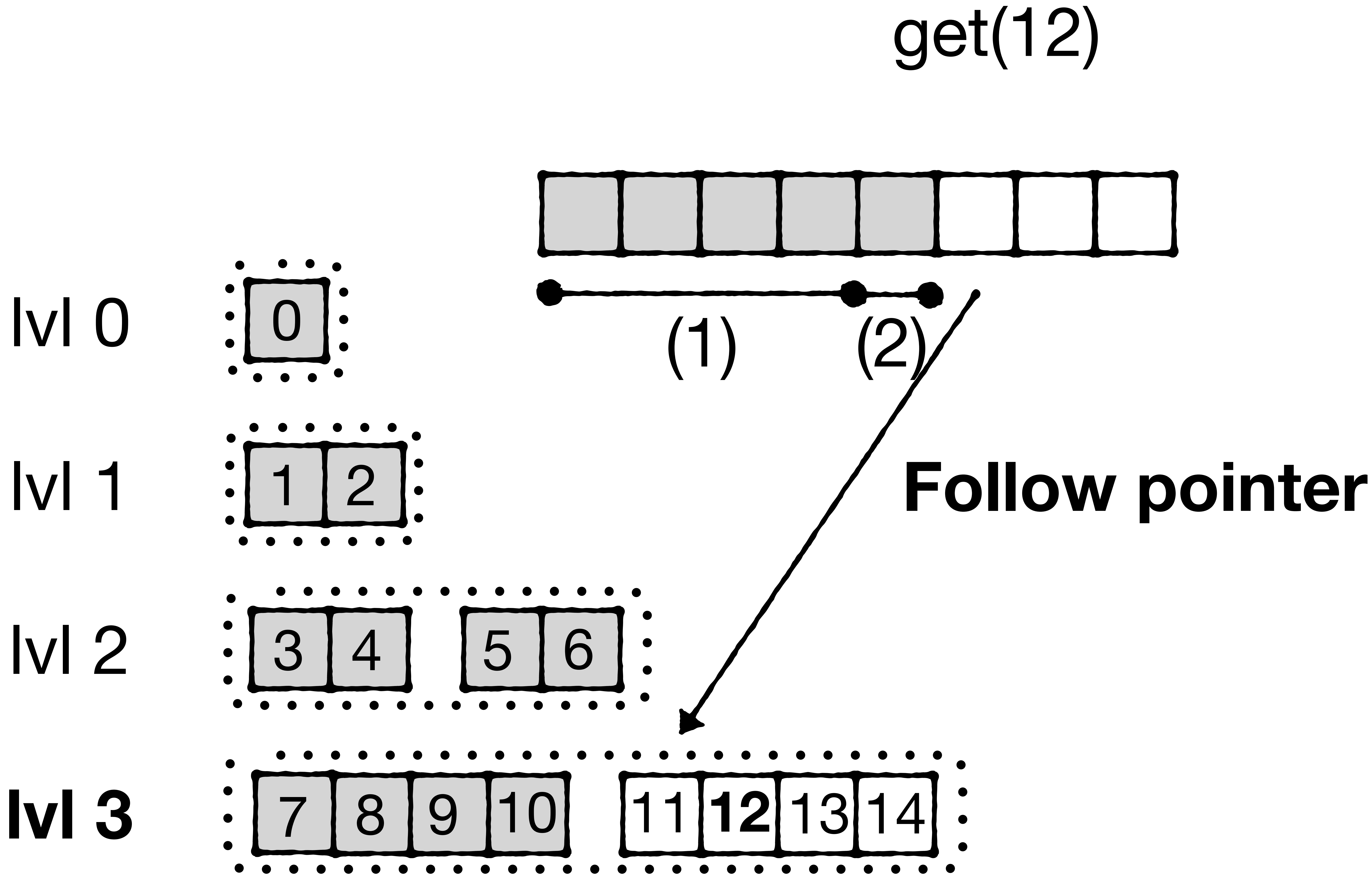


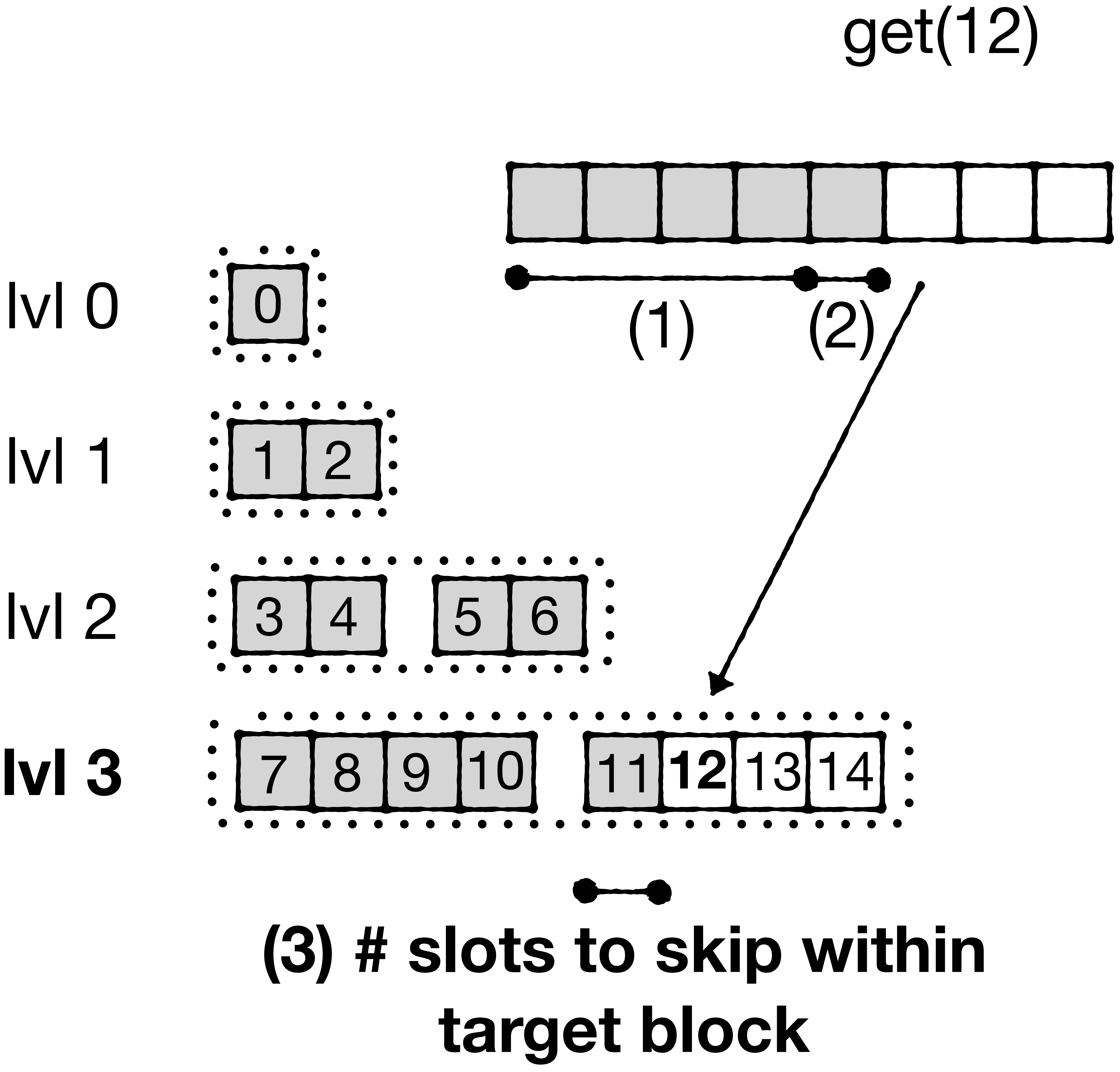
get(12)

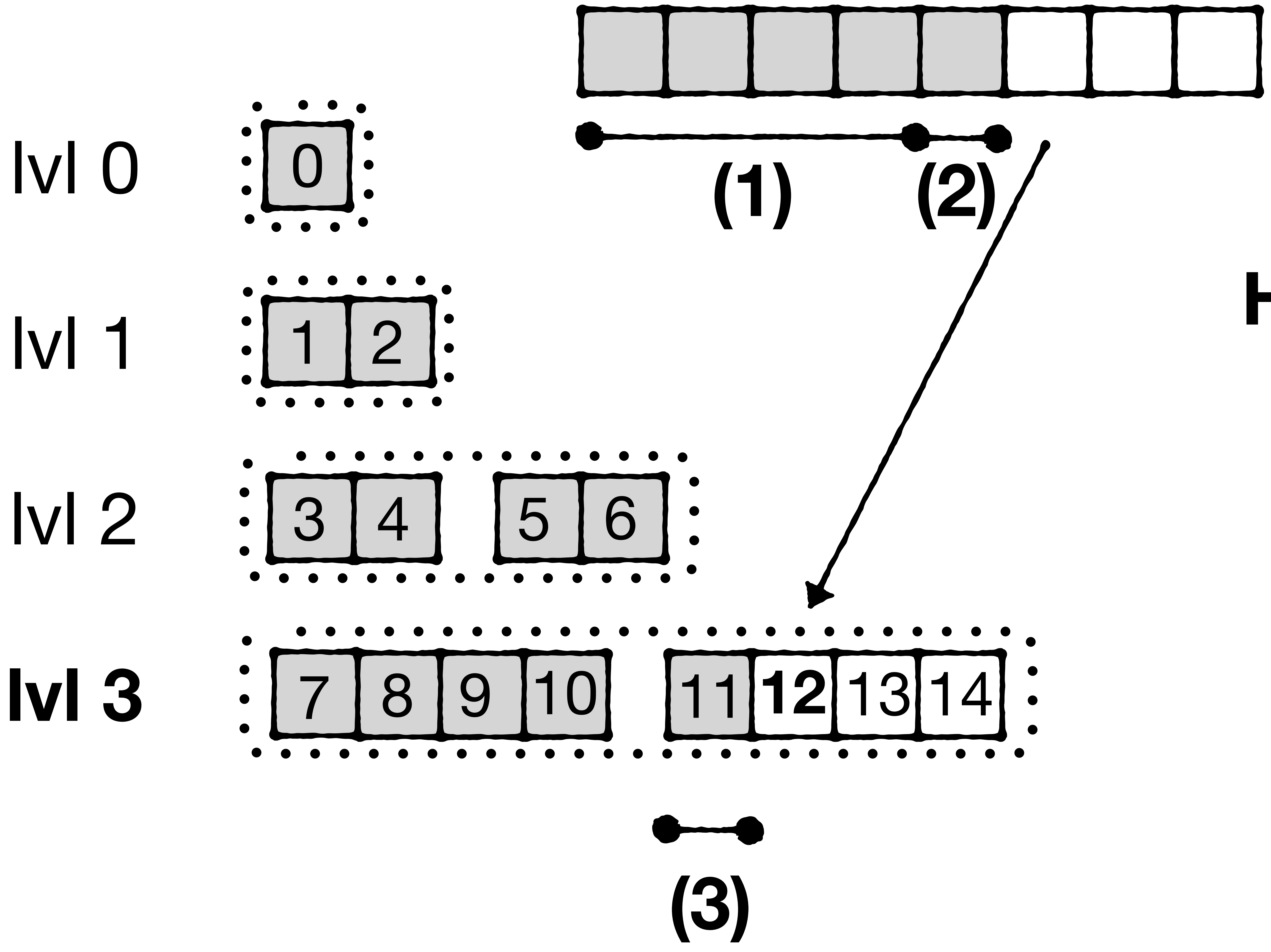


(1) # blocks to skip in smaller levels - tricky

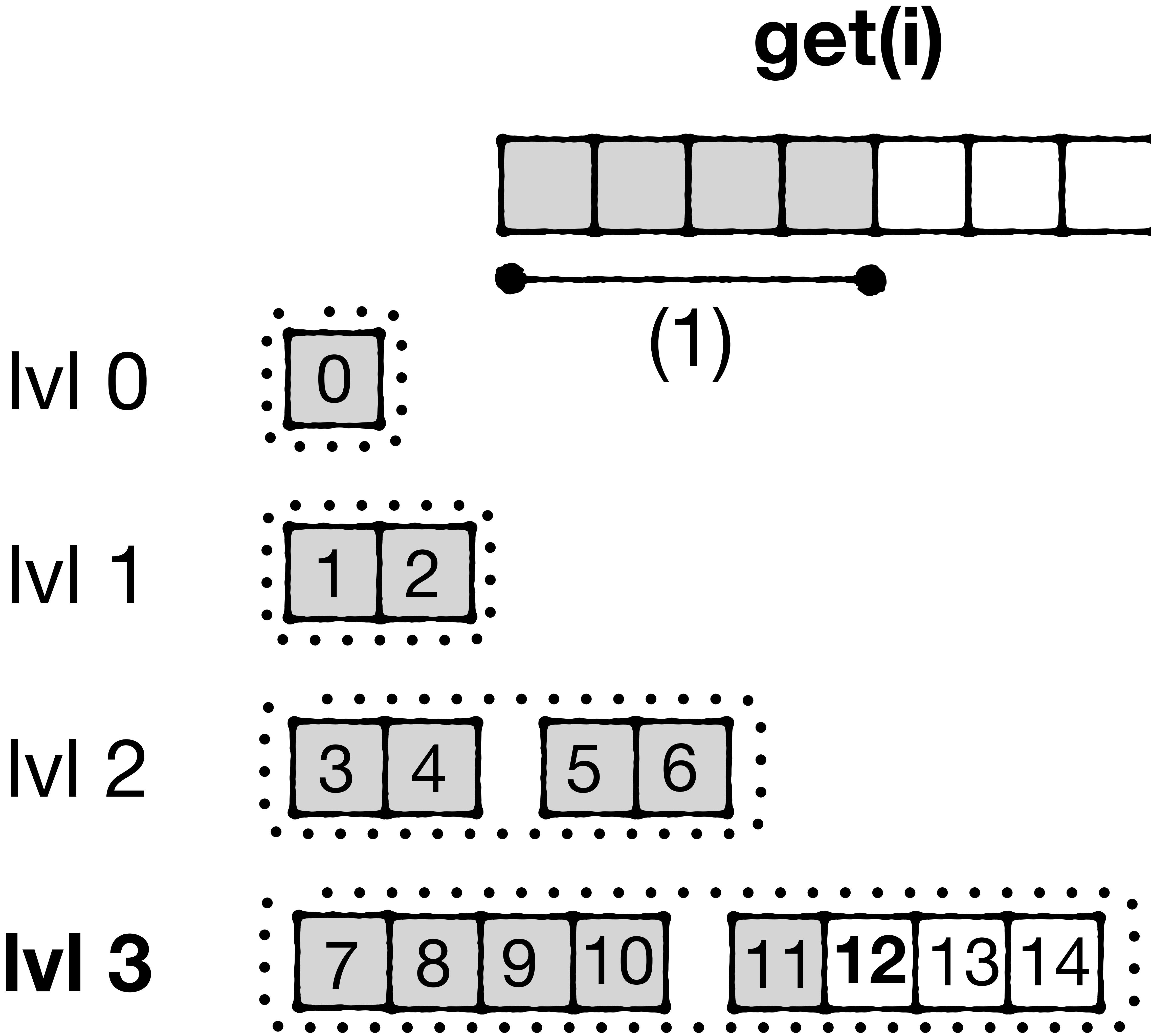

get(12)
lvl 0
0
(1)
(2) # blocks to skip in target level
lvl 1
1
2
lvl 2
3
4
5
6
lvl 3
7
8
9
10
11
12
13
14

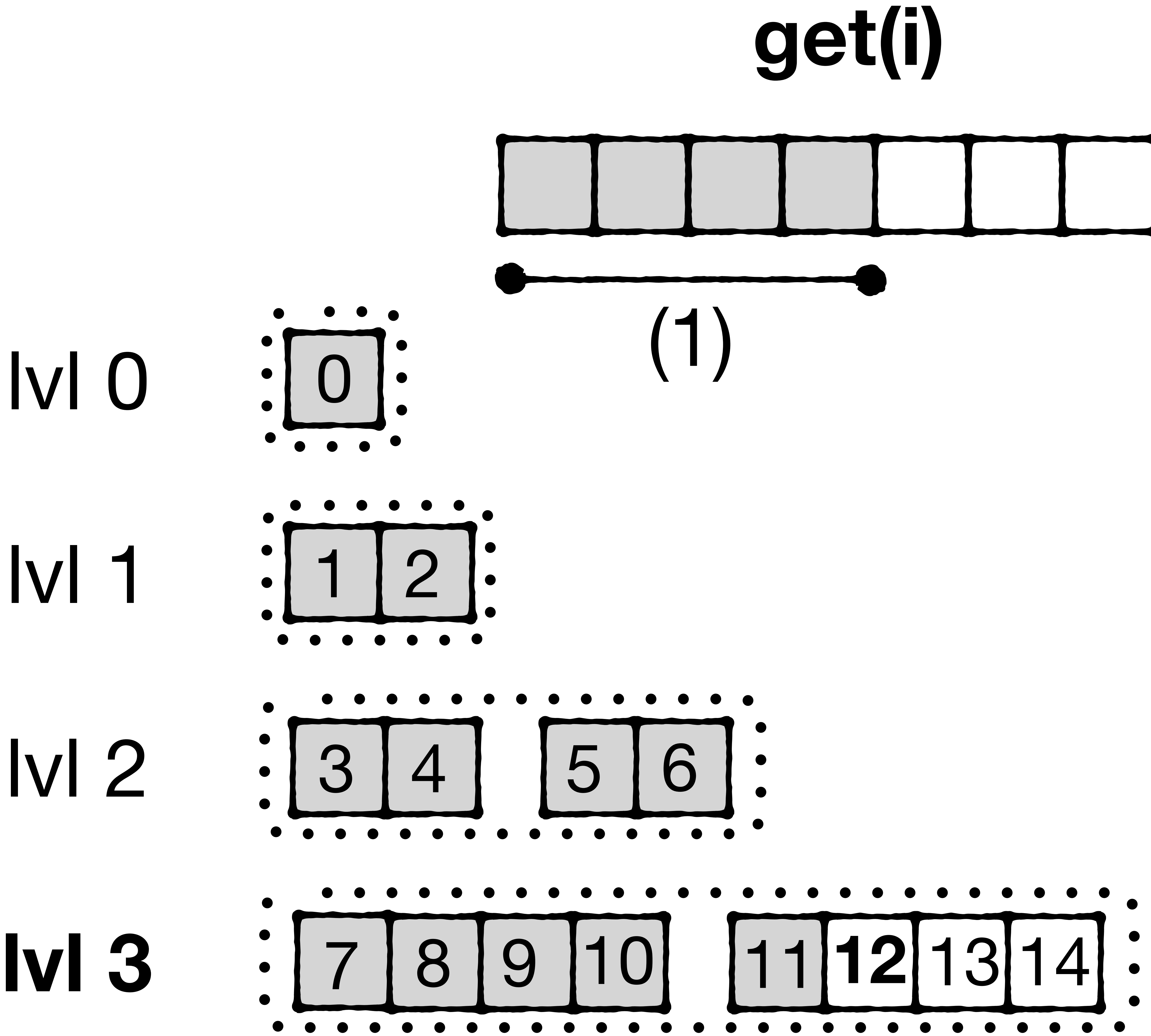

get(12)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(1)
(2)
Follow pointer


get(12)
lvl 0
0
(1)
(2)
lvl 1
1
2
lvl 2
3
4
5
6
lvl 3
7
8
9
10
11
12
13
14
(3) # slots to skip within target block



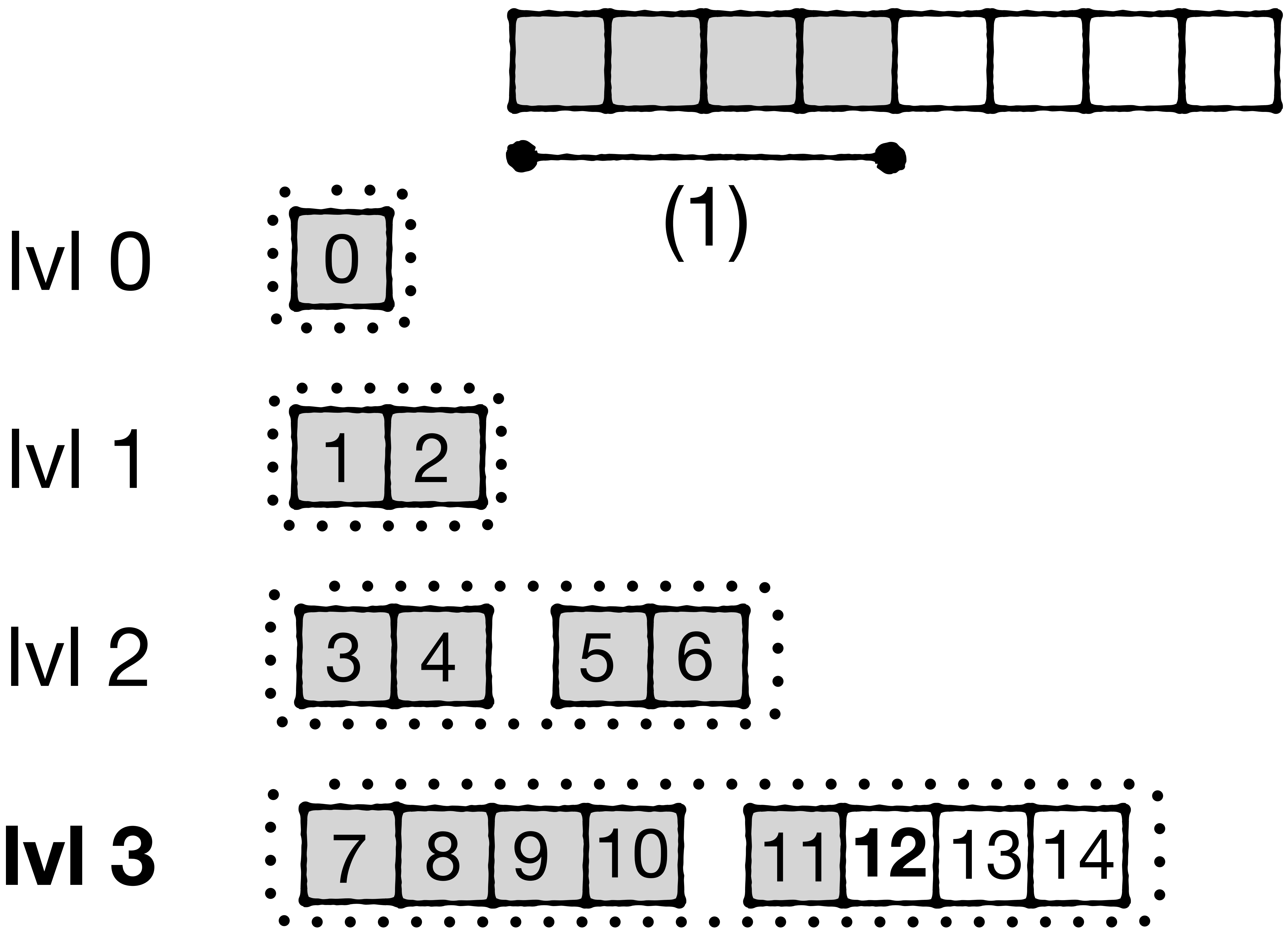
How to do steps 1 to 3 super fast?


get(i)
(1)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14



Identify target level k

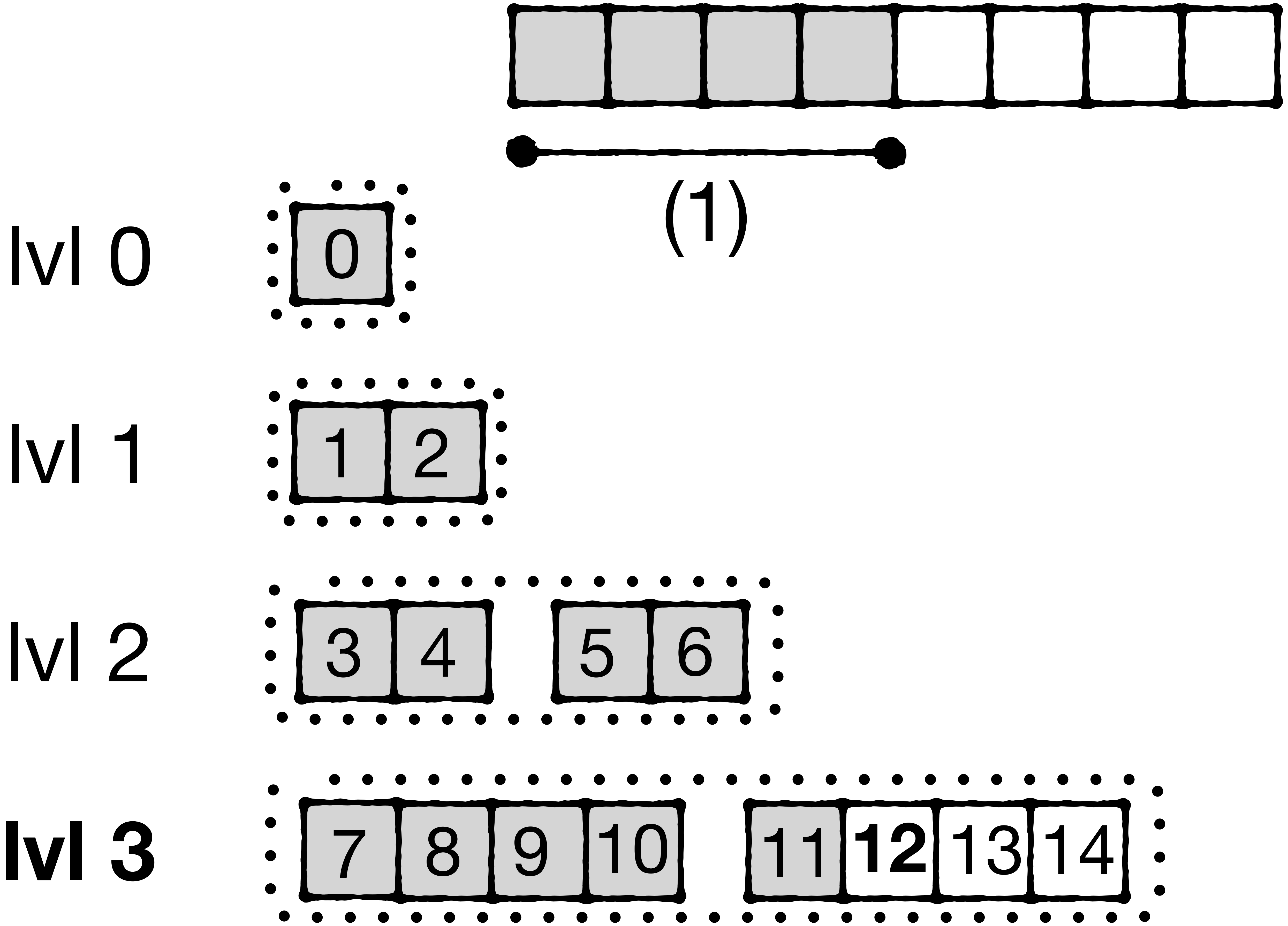
get(i)



Identify target level k

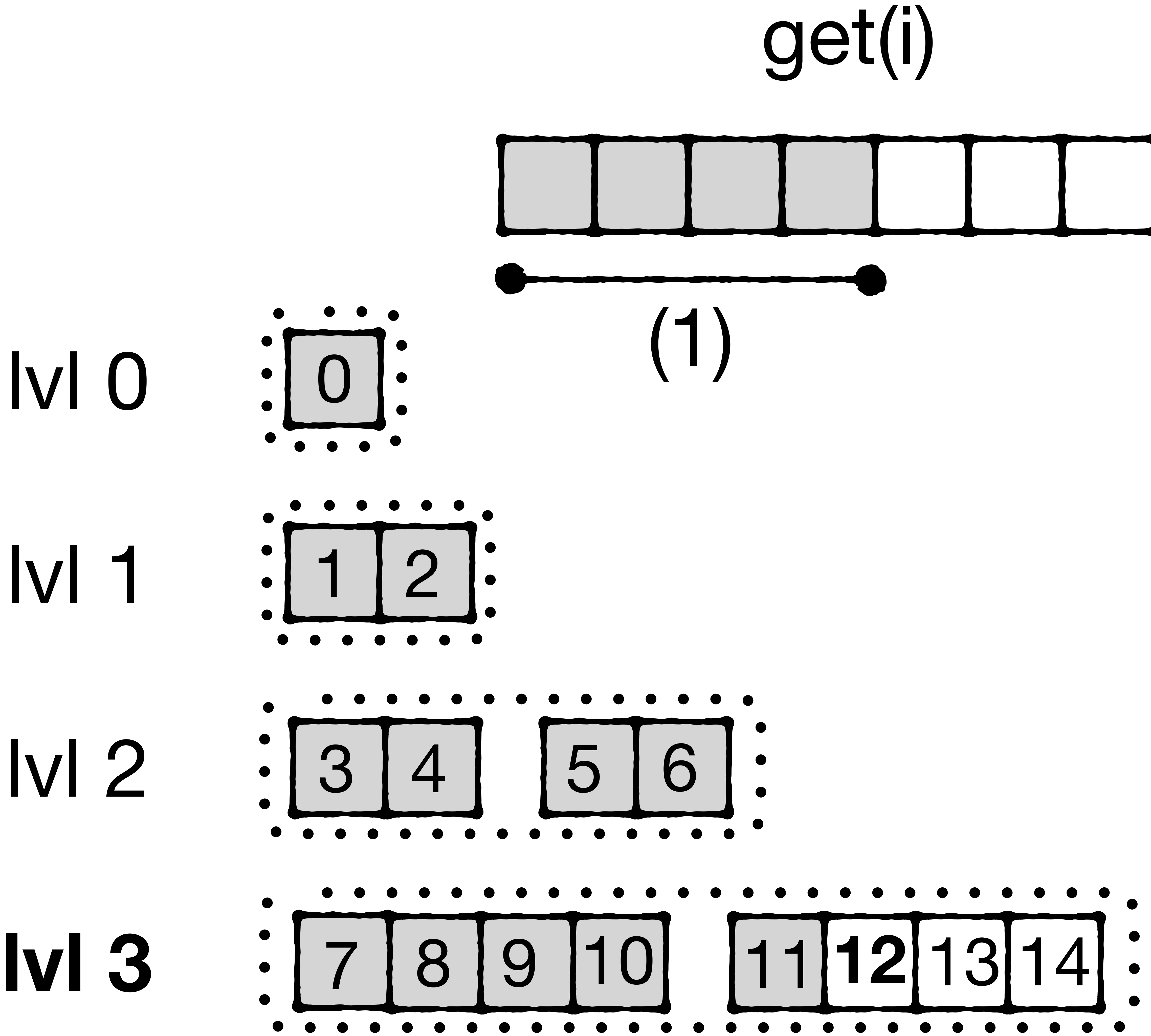
$$k = \lfloor \log_2(i + 1) \rfloor$$

get(12)



Identify target level k

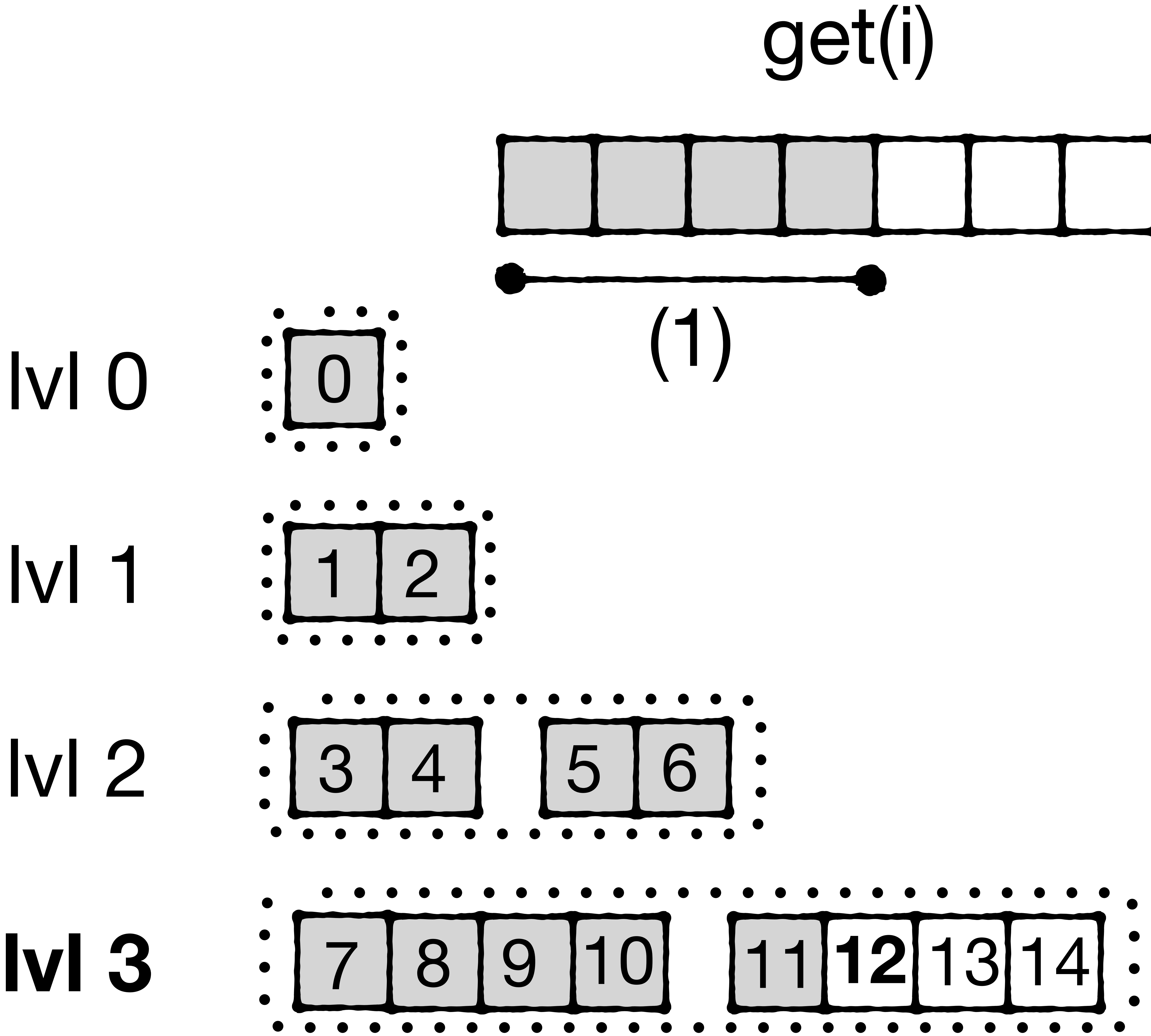
$$k = \lfloor \log_2(12 + 1) \rfloor = 3$$



Identify target level k

$k = \lfloor \mathbf{log_2}(i + 1) \rfloor$

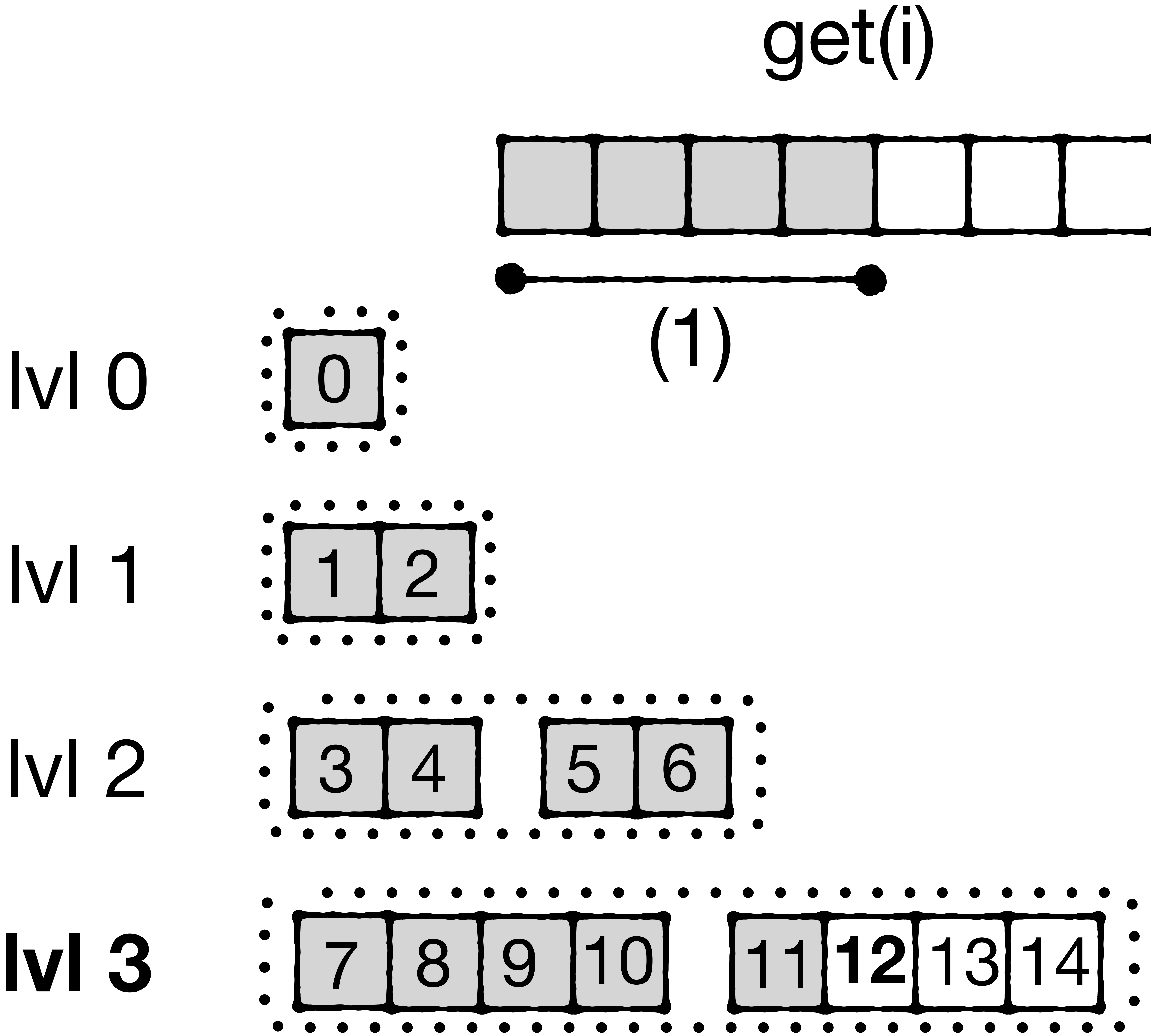
slow



Identify target level k

$$k = \lfloor \log_2(i + 1) \rfloor$$

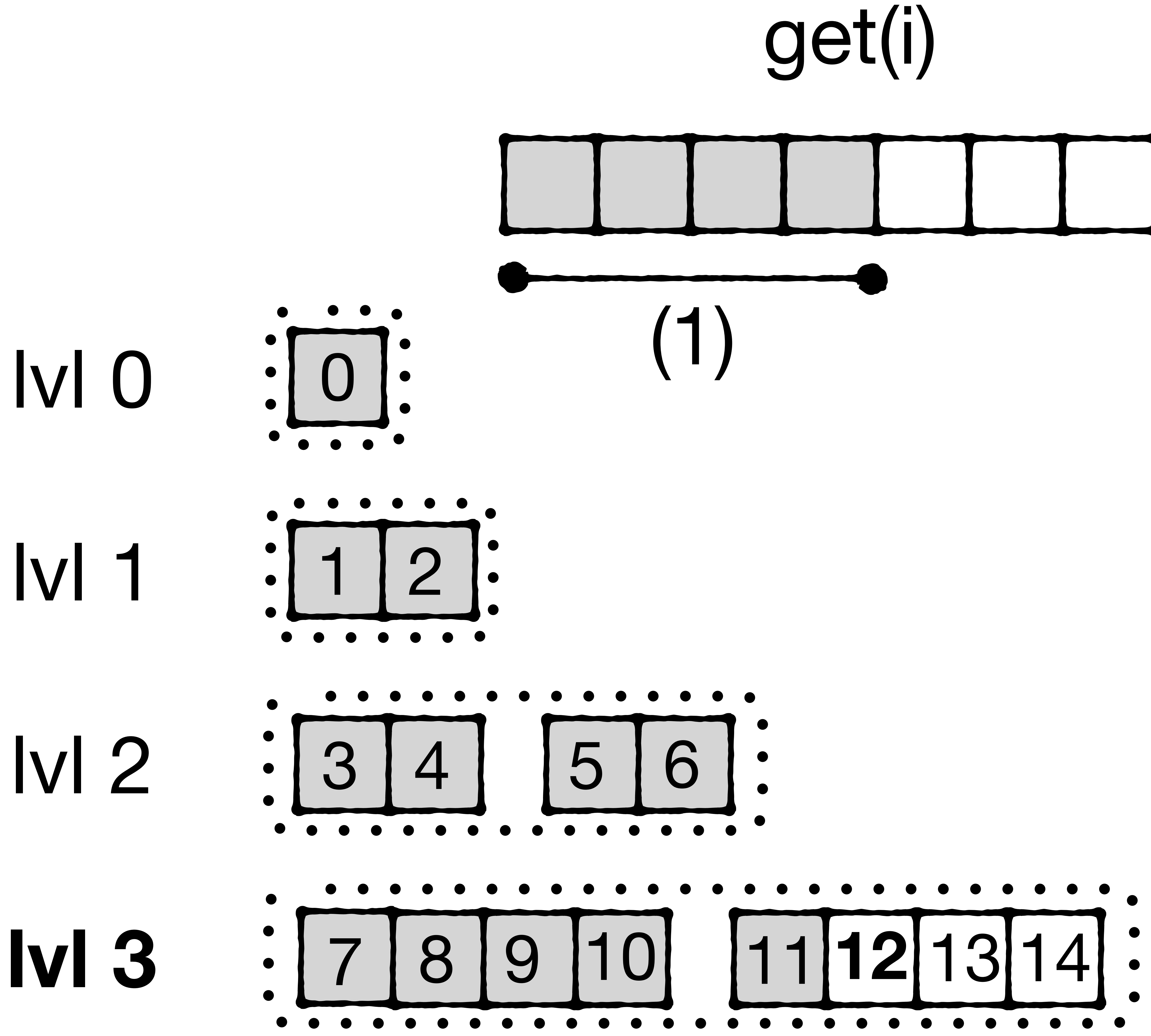
Type casting - also slow



Identify target level k

$$k = \lfloor \log_2(i + 1) \rfloor$$

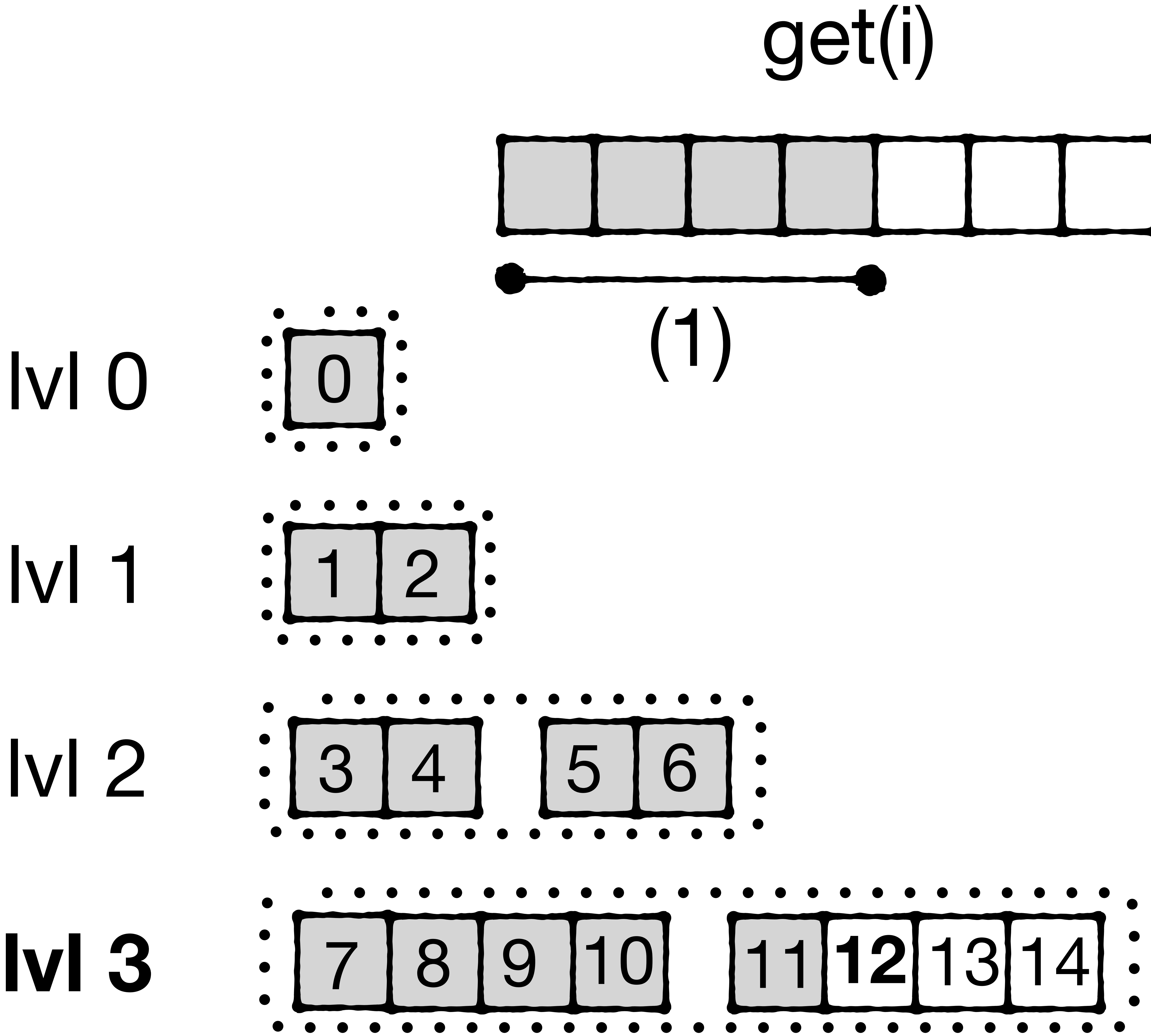
Insight?



Identify target level k

$k = \lfloor \log_2(i + 1) \rfloor$

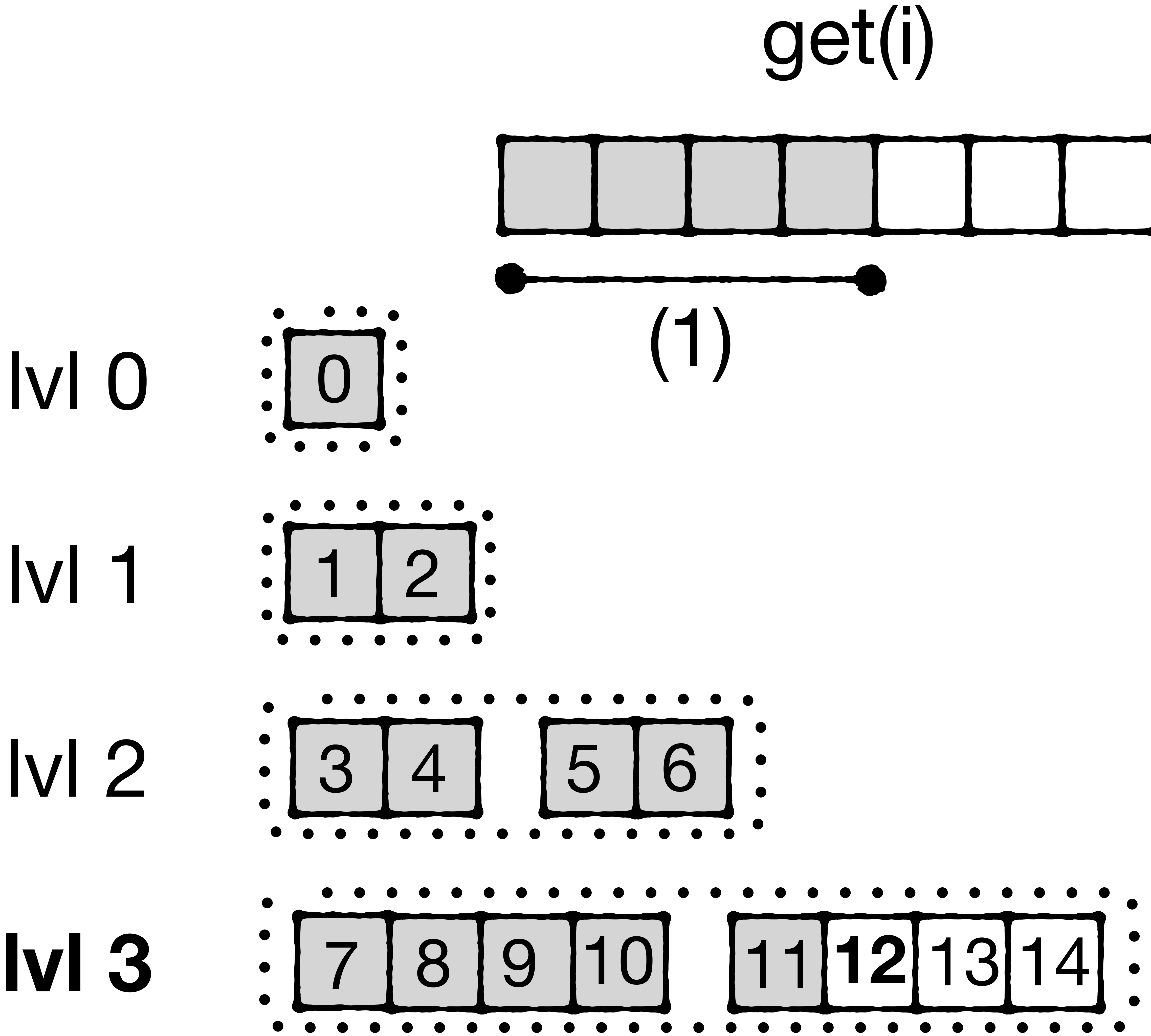
Insight: $\log_2$ amounts to finding index of most significant digit



Identify target level k

$$k = \lfloor \log_2(i + 1) \rfloor$$

$$= \mathbf{sizeof(i) - 1 - clz(i+1)}$$



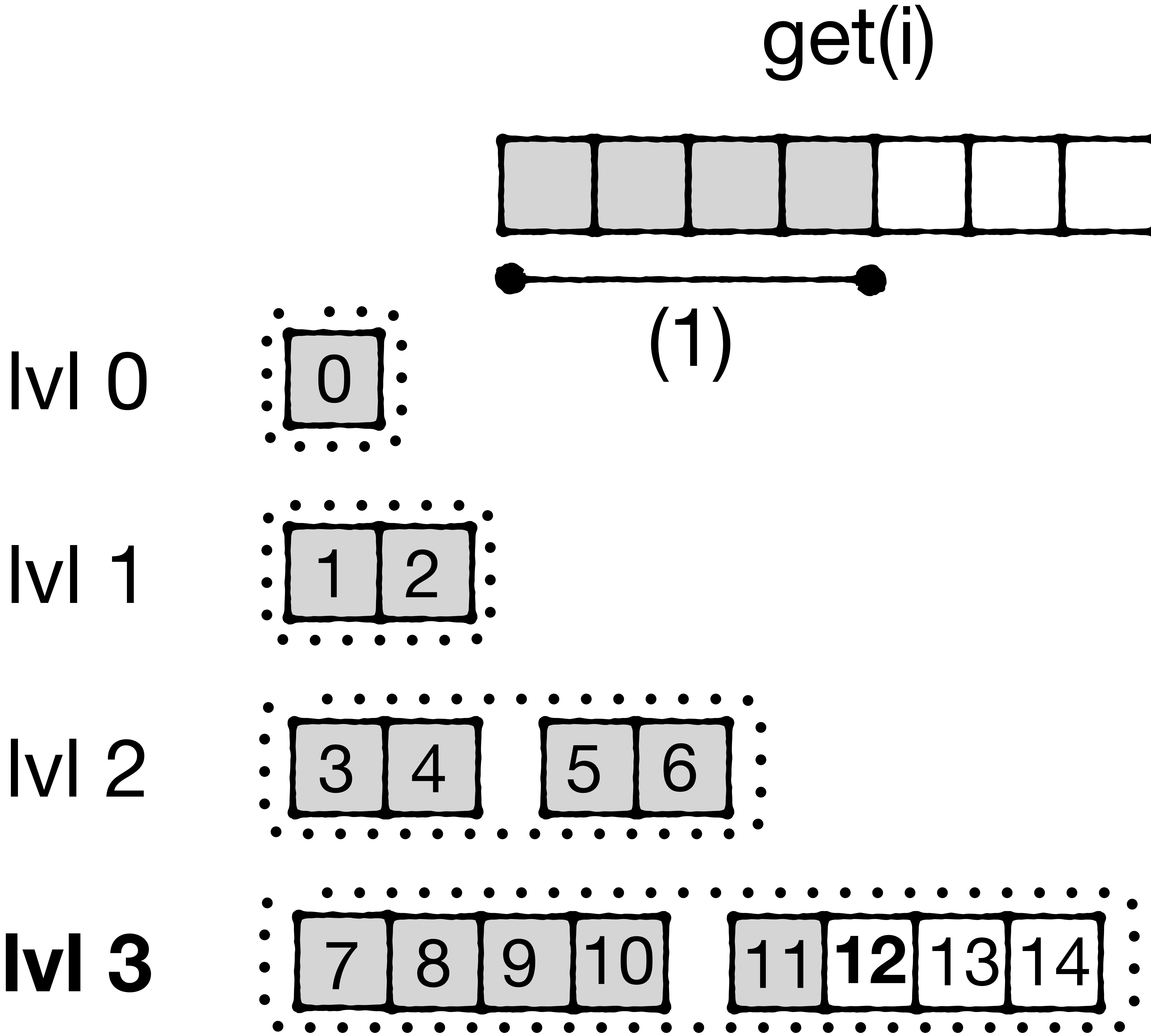
Identify target level k

$$k = \lfloor \log_2(i + 1) \rfloor$$

$= \textbf{sizeof(i)} - 1 - \mathrm{clz}(i+1)$

↑

Integer length in bits



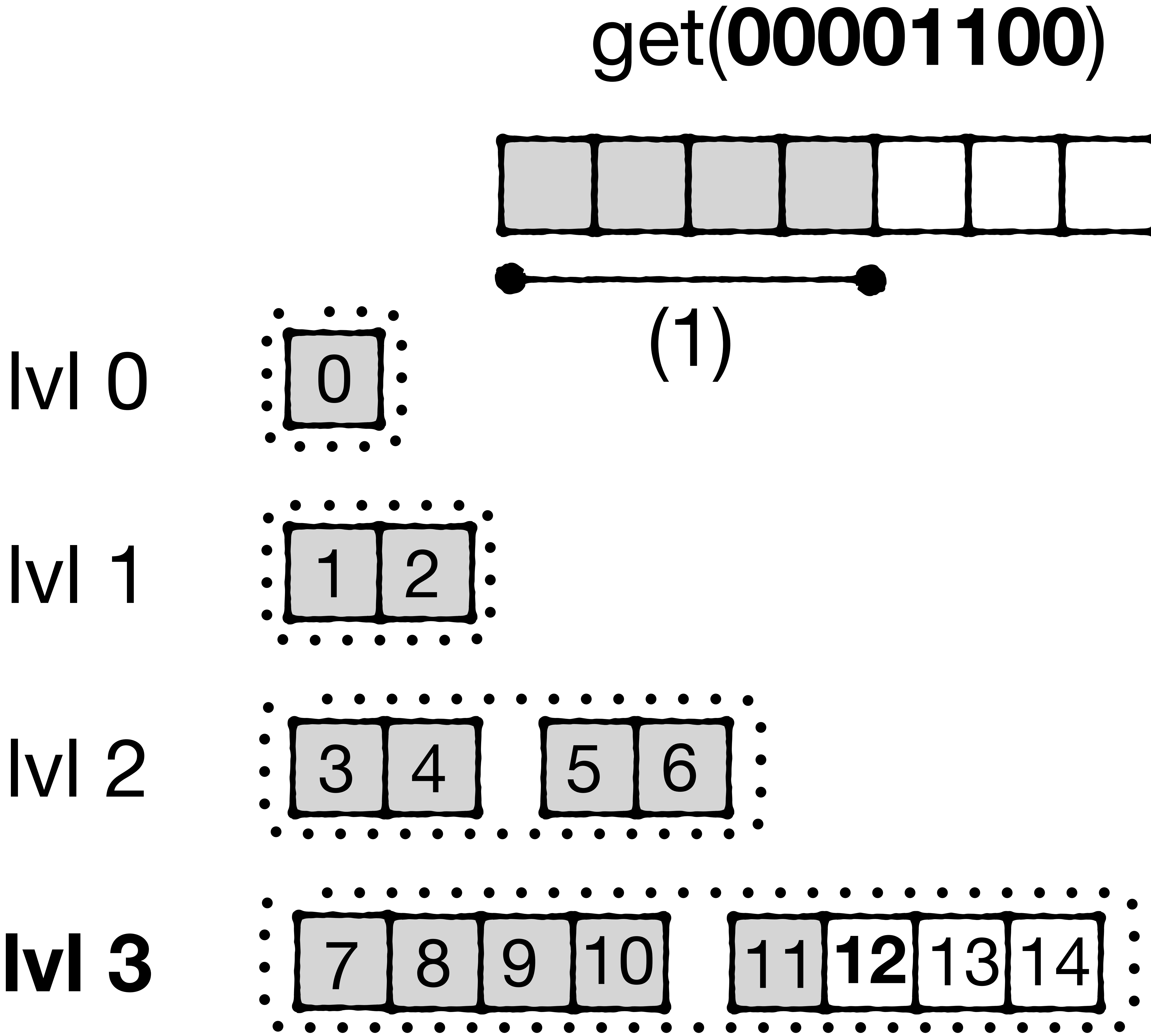
Identify target level k

$$k = \lfloor \log_2(i+1) \rfloor$$

$$= \text{sizeof}(i) - 1 - \mathbf{clz(i+1)}$$

↑

Specialized CPU command for # leading zeros

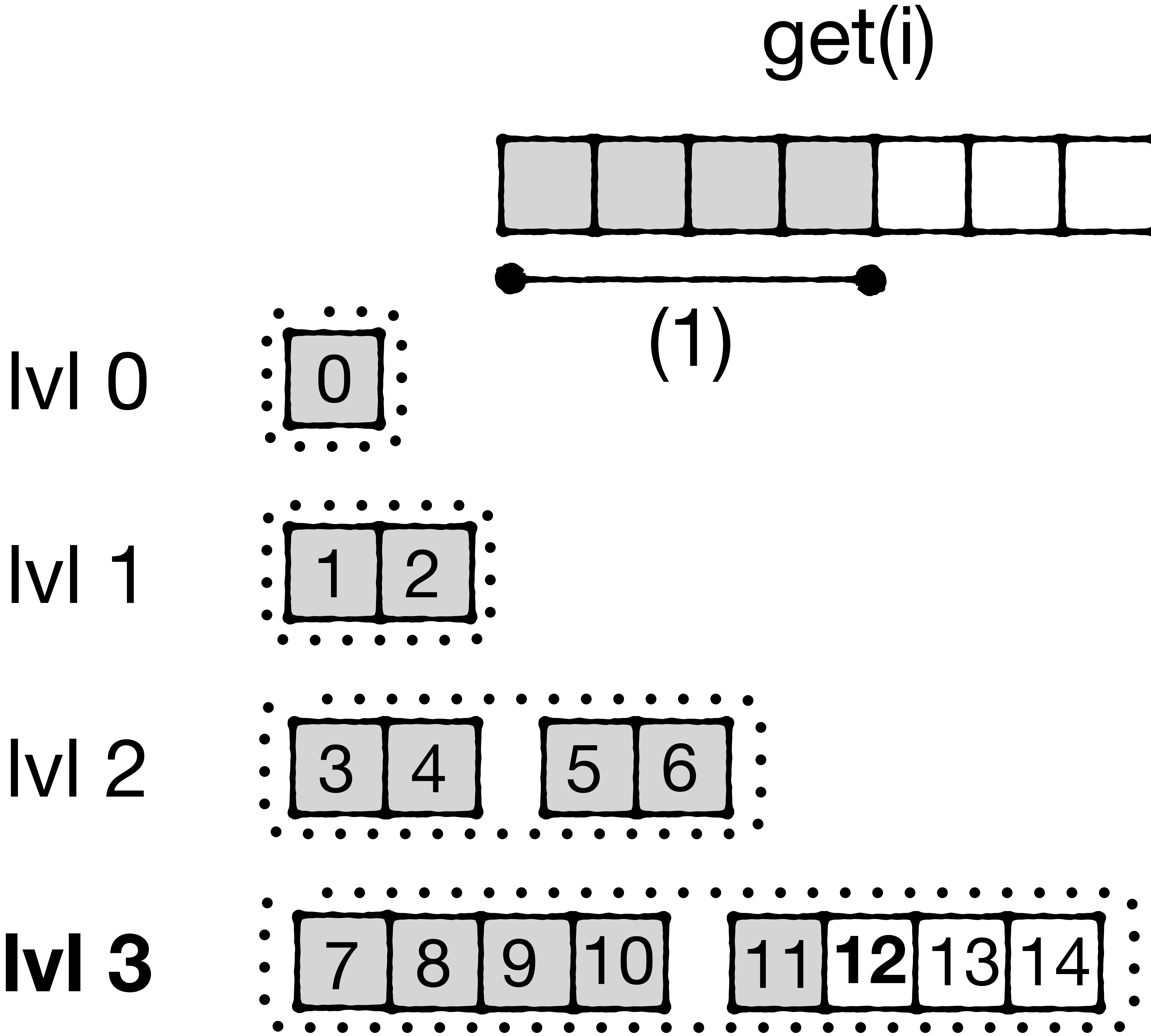


Identify target level k

$$k = \lfloor \log_2(i + 1) \rfloor$$

$$= \text{sizeof}(i) - 1 - \text{clz}(i+1)$$

$$\mathbf{8 \quad - 1 - 4 = 3}$$

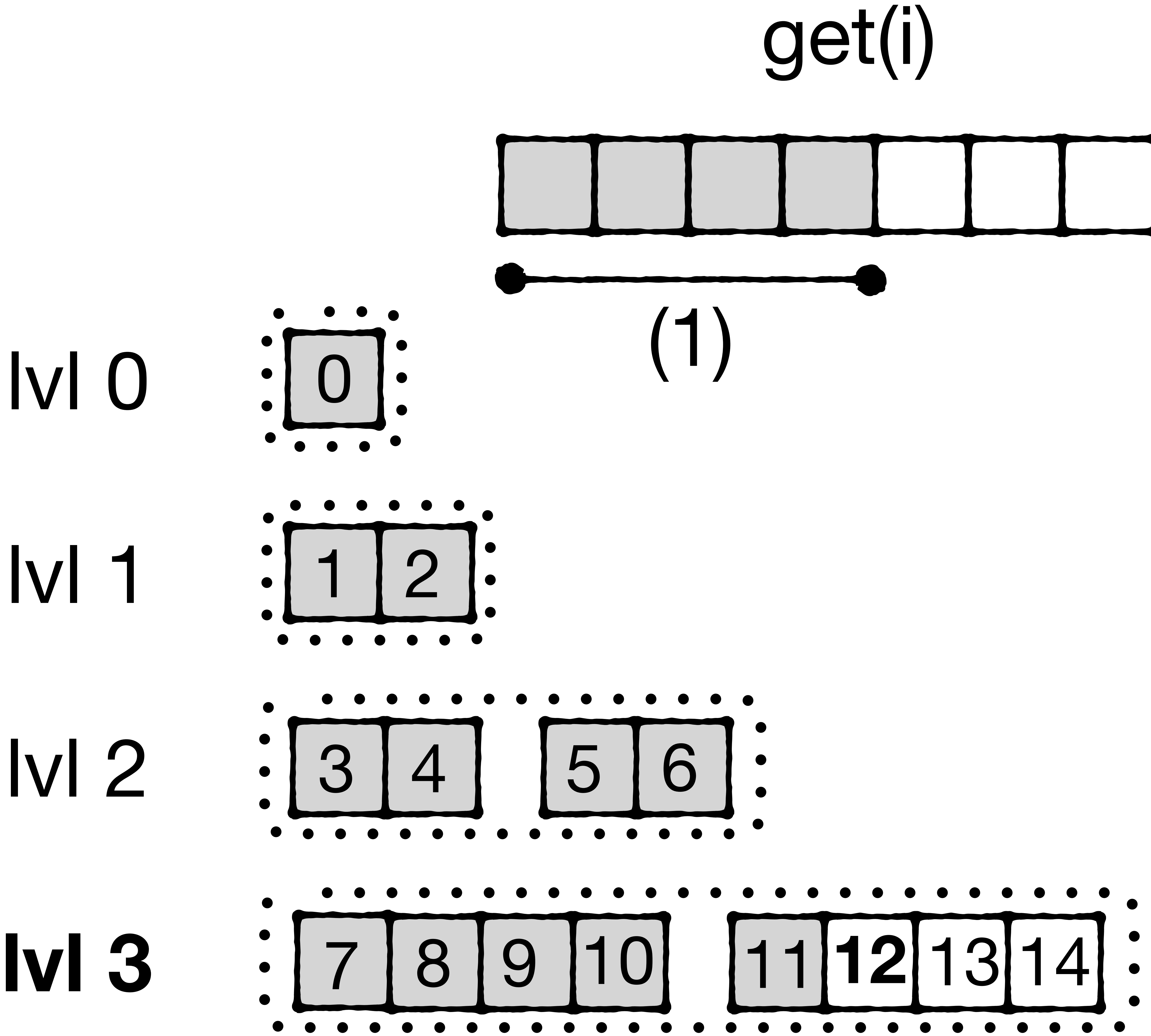


Identify target level k

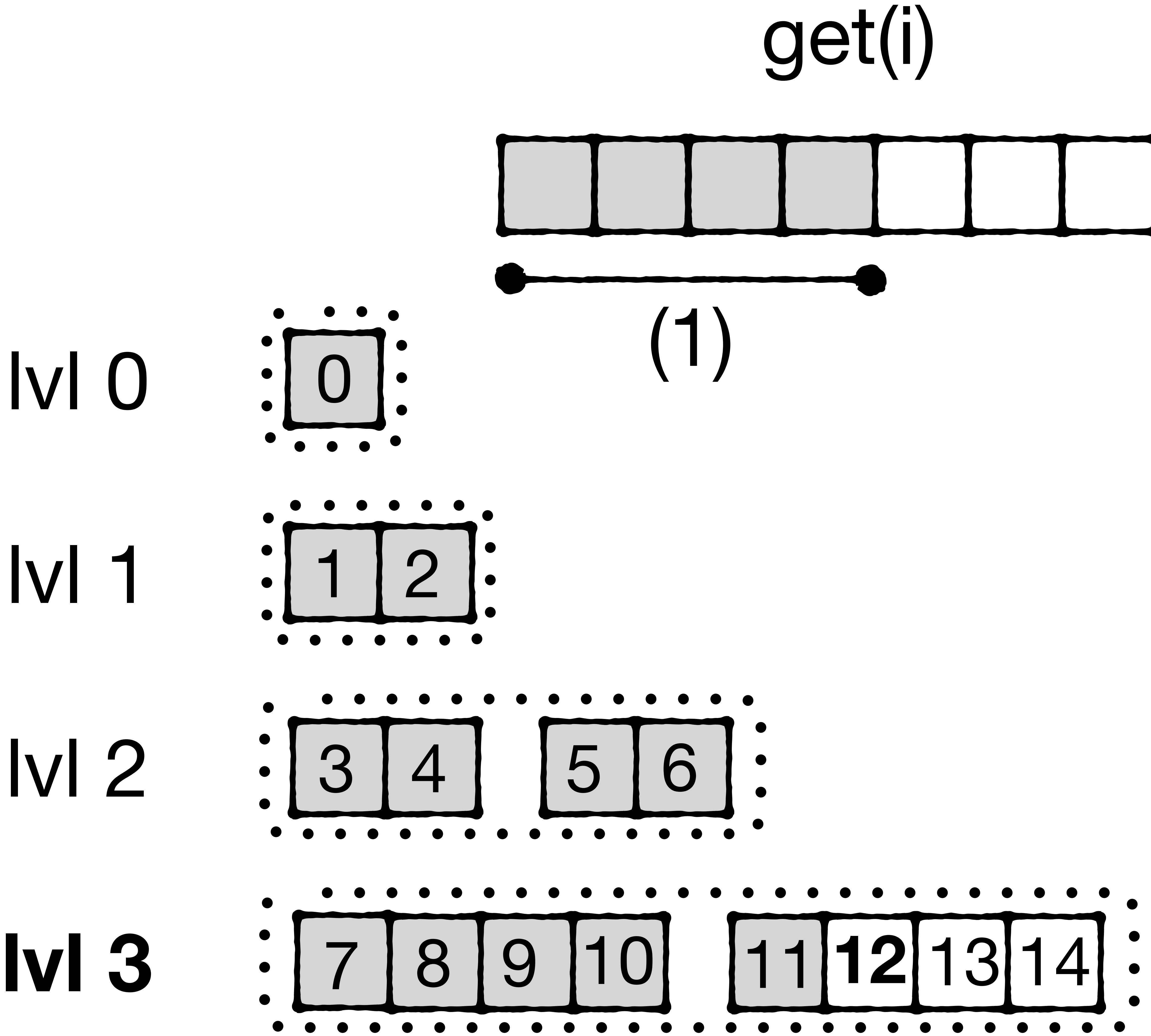
$$k = \lfloor \log_2(i + 1) \rfloor$$

$$= \text{sizeof}(i) - 1 - \text{clz}(i+1)$$

≈1 ns rather than ≈7ns

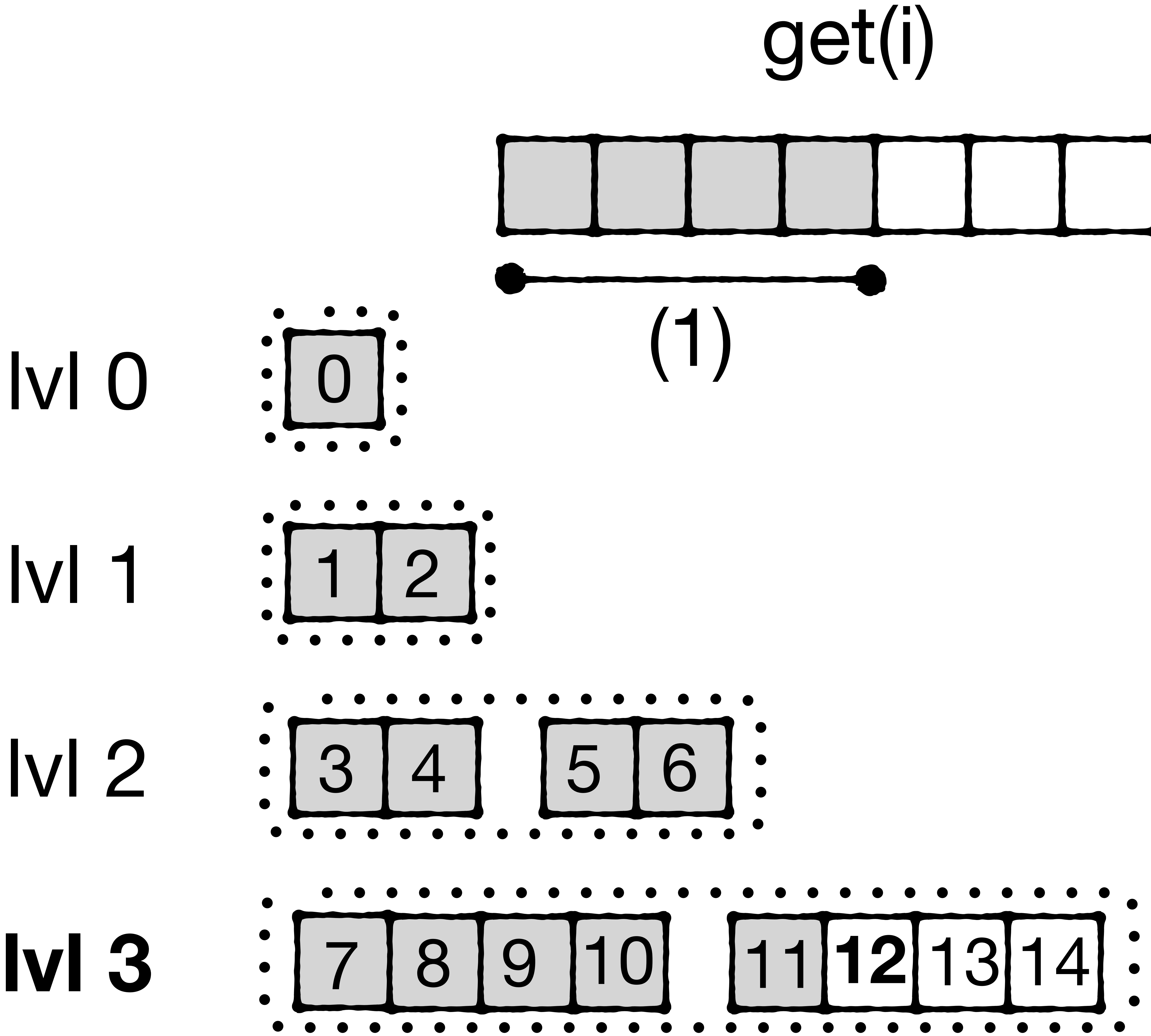


 # blocks in levels 0 to k-1?



 # blocks in levels 0 to k-1

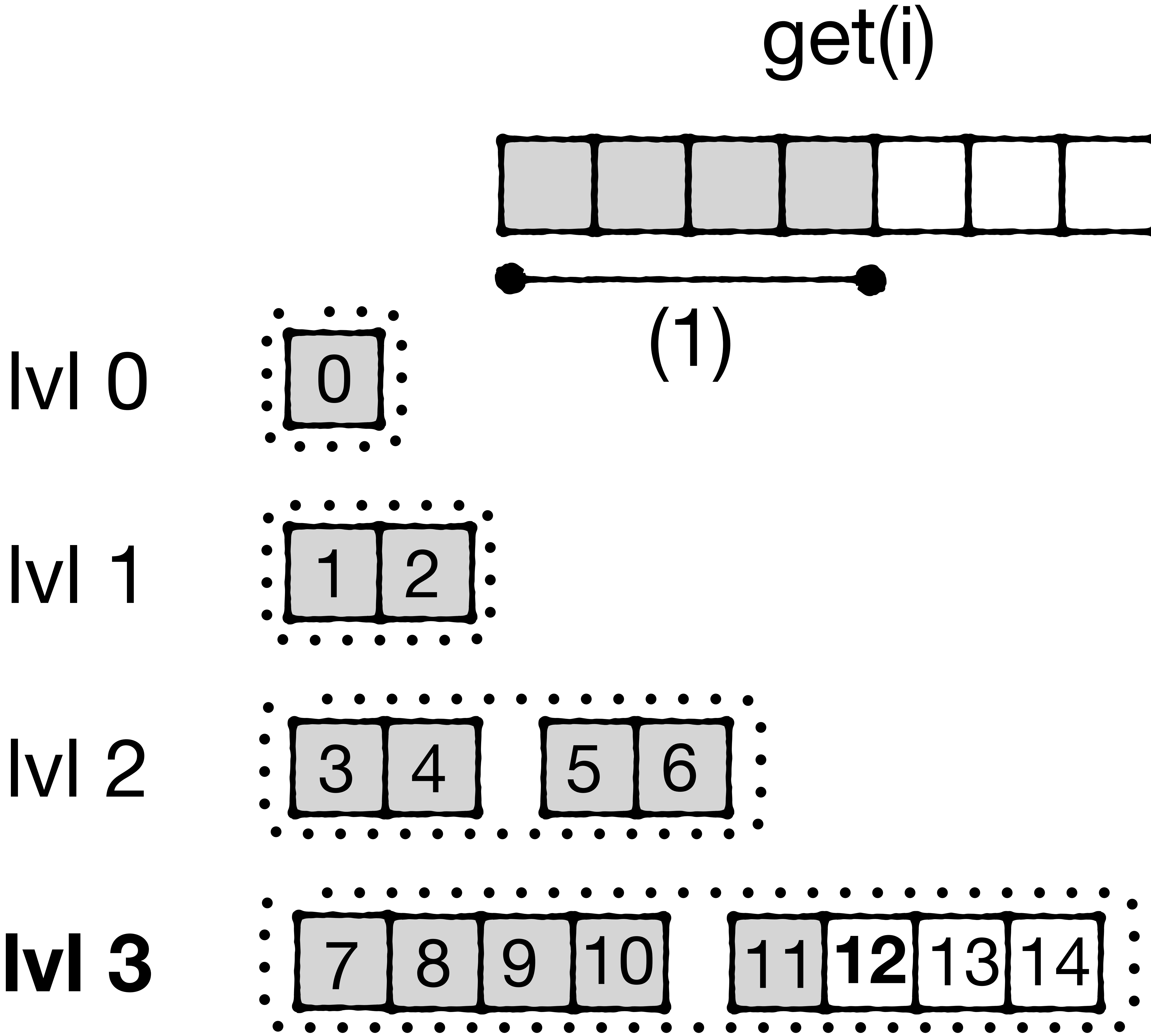
$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$



 # blocks in levels 0 to k-1

$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

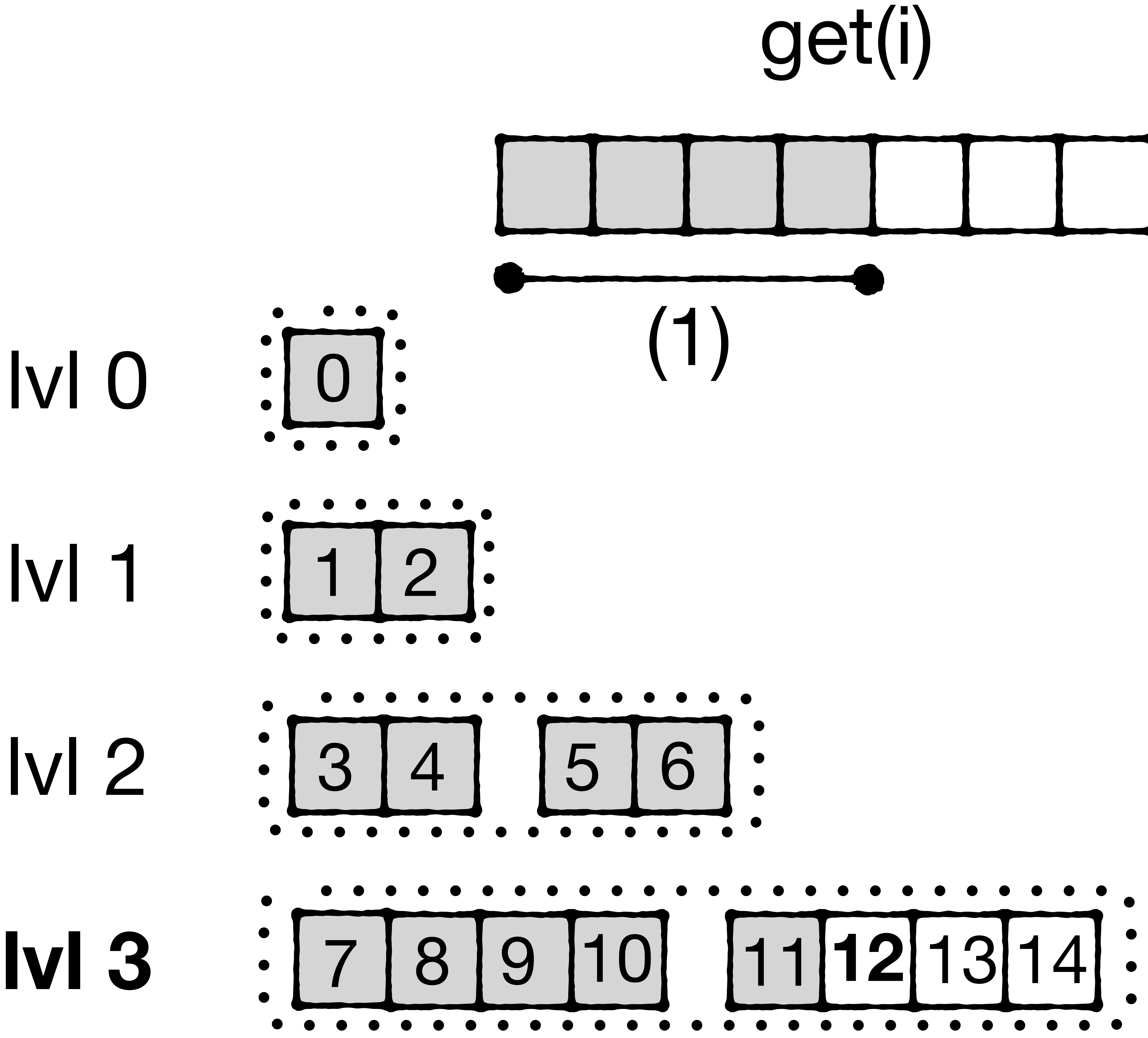
Original paper gets this wrong, fixed credit to Hyuhng Min



\# blocks in levels 0 to k-1

$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (\mathbf{k \bmod 2})) - 2$

Intuition: number of new data blocks grows every other level

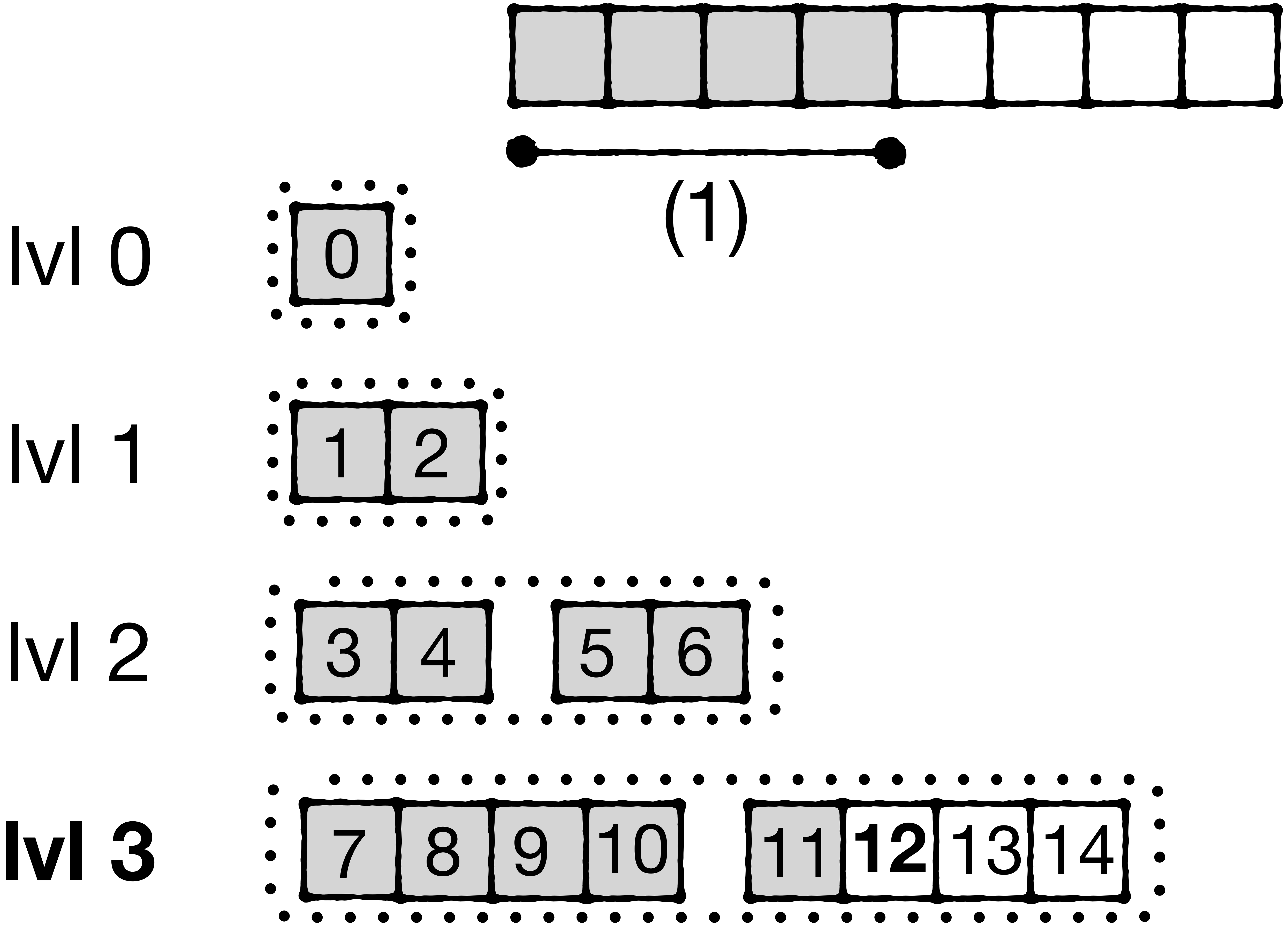


blocks in levels 0 to k-1

$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

Level k	# Blocks
0	0
1	1
2	2
3	4
4	6
5	10
6	14
7	22
...	...

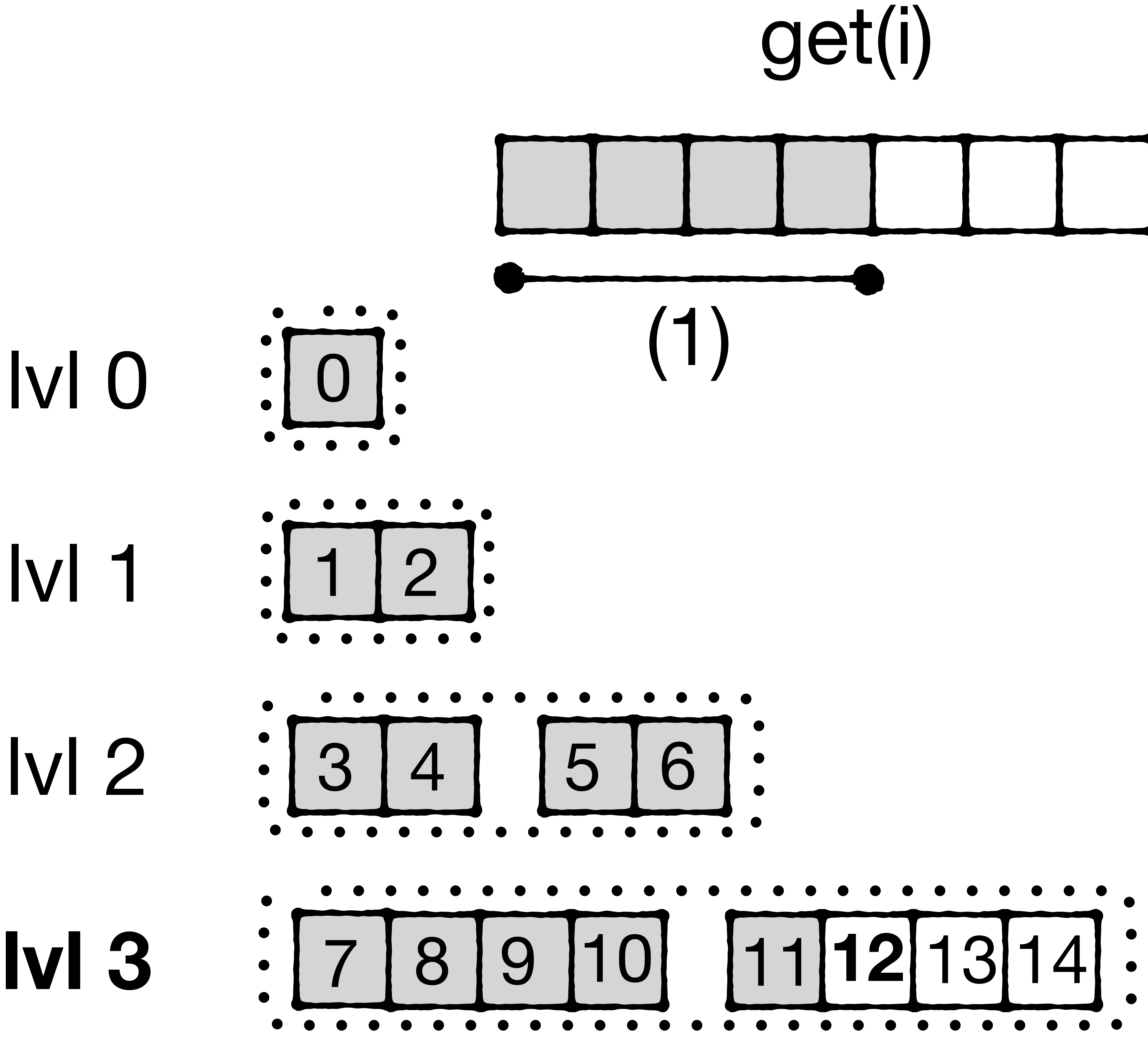
get(i)



blocks in levels 0 to k-1

$$= 2^{\lfloor \boldsymbol{k/2} \rfloor} \cdot (2 + (\mathbf{k \bmod 2})) - 2$$

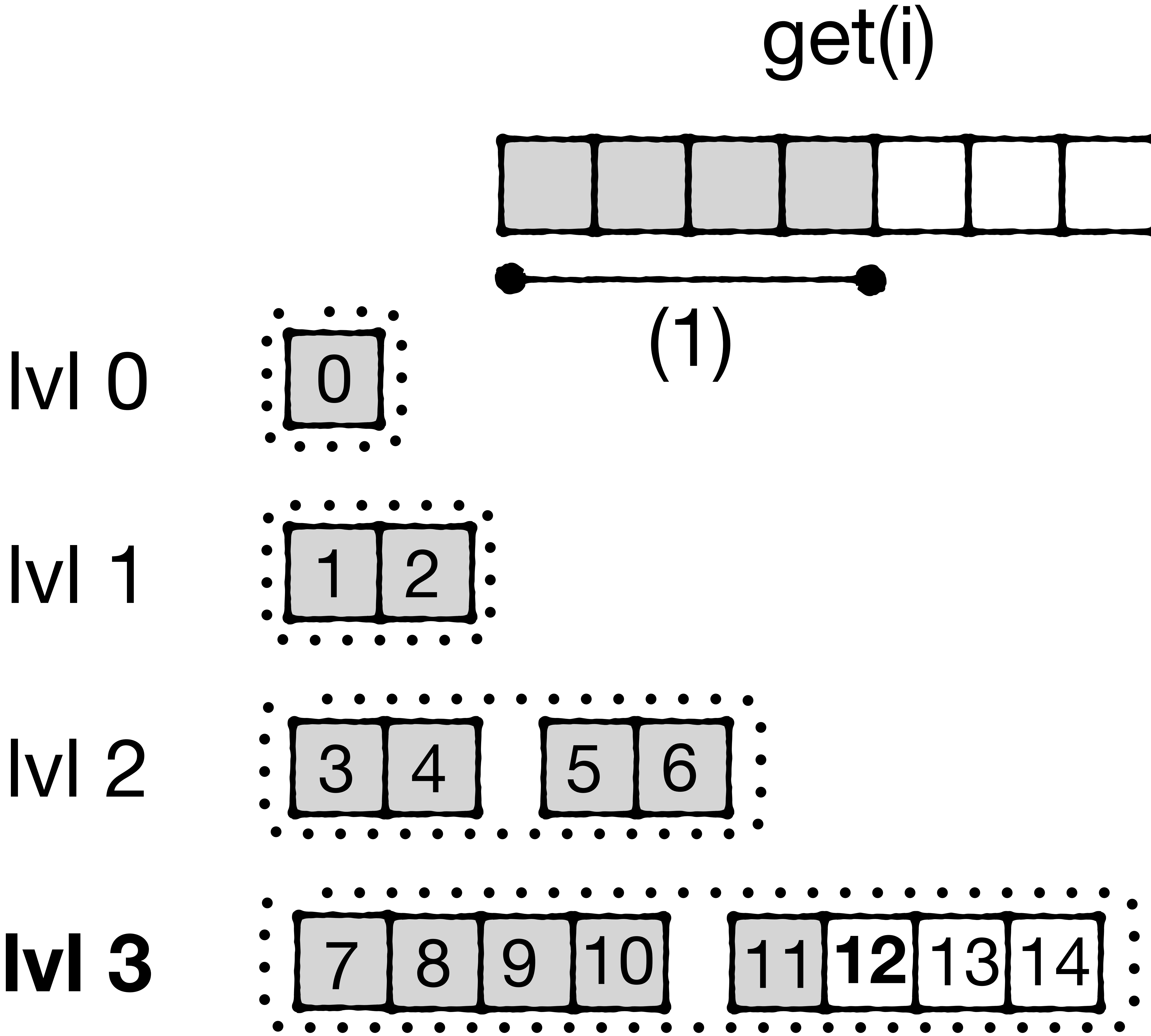
Integer division is slow



blocks in levels 0 to k-1

$$= \mathbf{2}^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

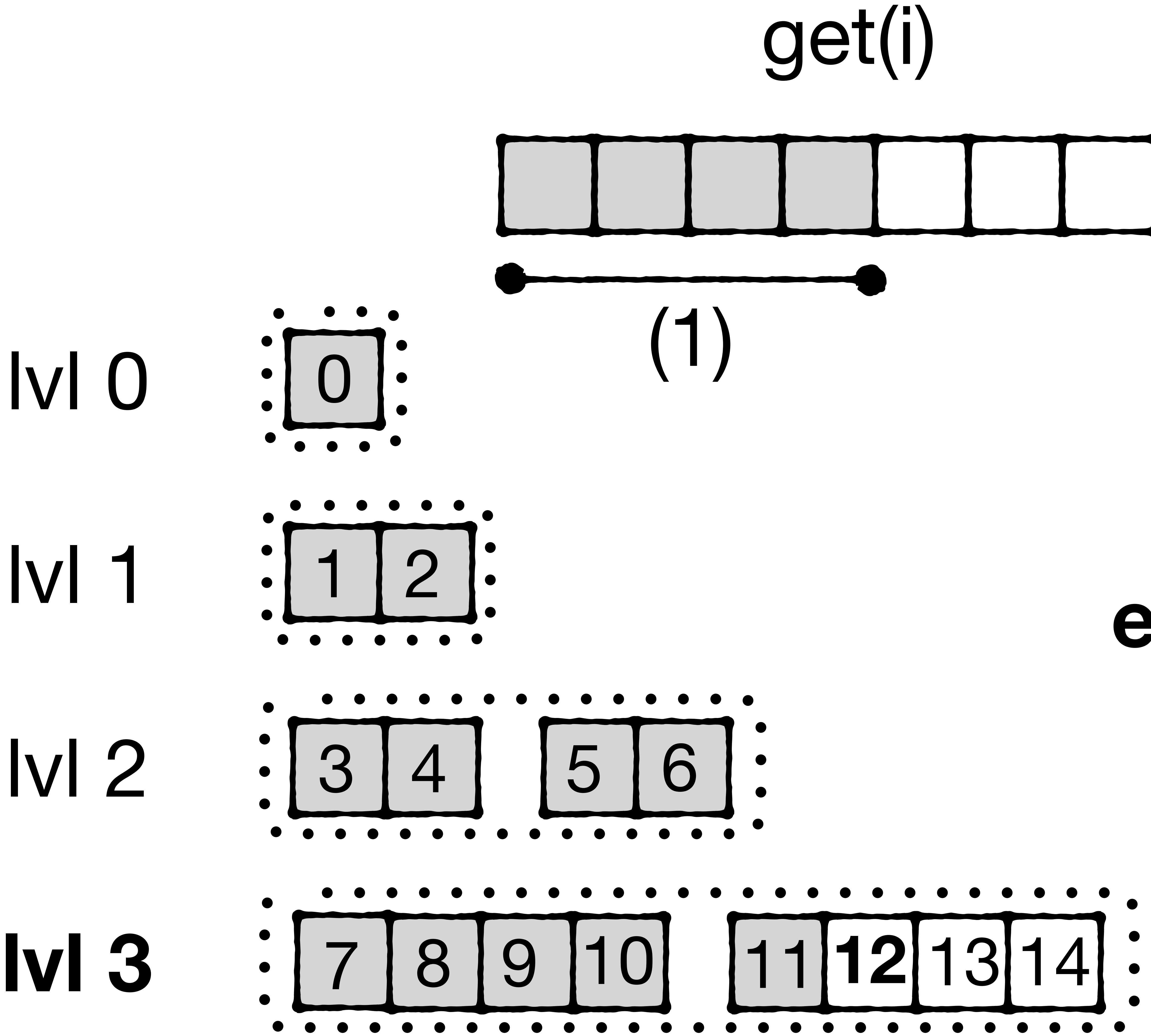
Power is slow



blocks in levels 0 to k-1

$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

How to speed up?

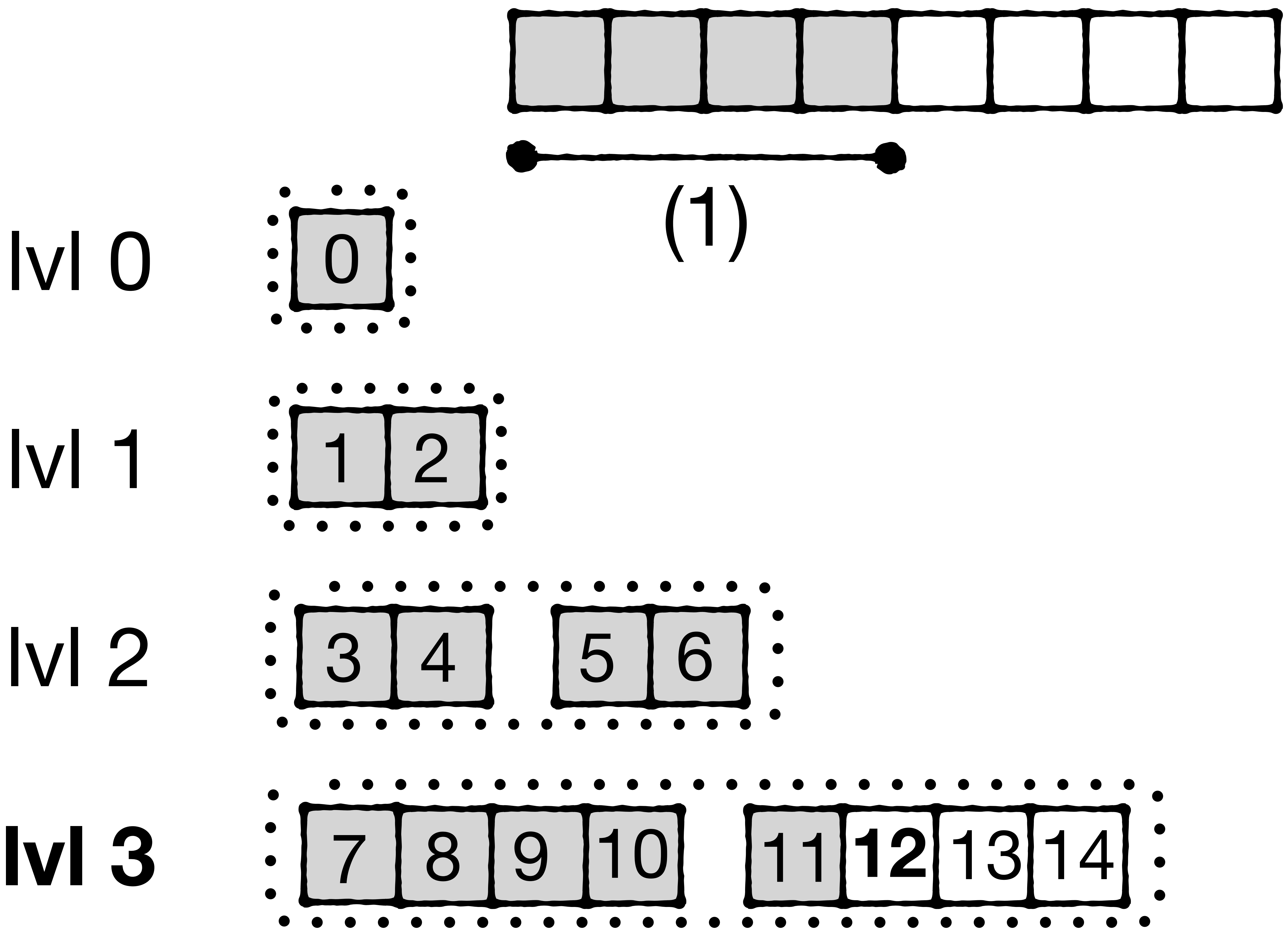


blocks in levels 0 to k-1

$= \mathbf{2}^{\lfloor k/2 \rfloor} \cdot (2 + (\mathbf{k\ mod\ 2})) - 2$

Insight: division & exponentiation by 2 can be done with bitwise operators

get(i)

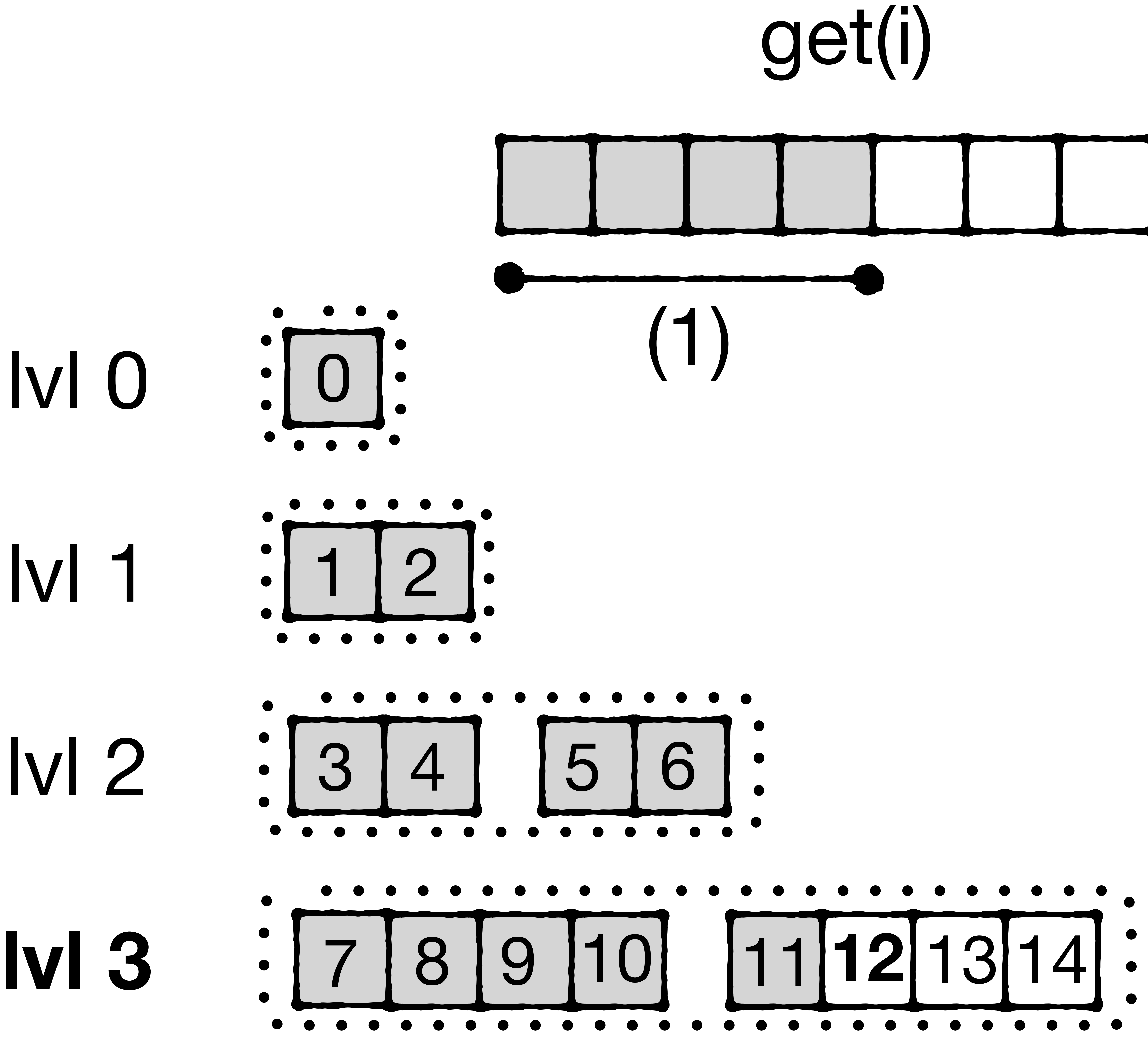


blocks in levels 0 to k-1

$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$

$= 2^{\lfloor \boldsymbol{k/2} \rfloor} \cdot (2 + (\mathbf{k\ \&\ 1})) - 2$

“and” with 1



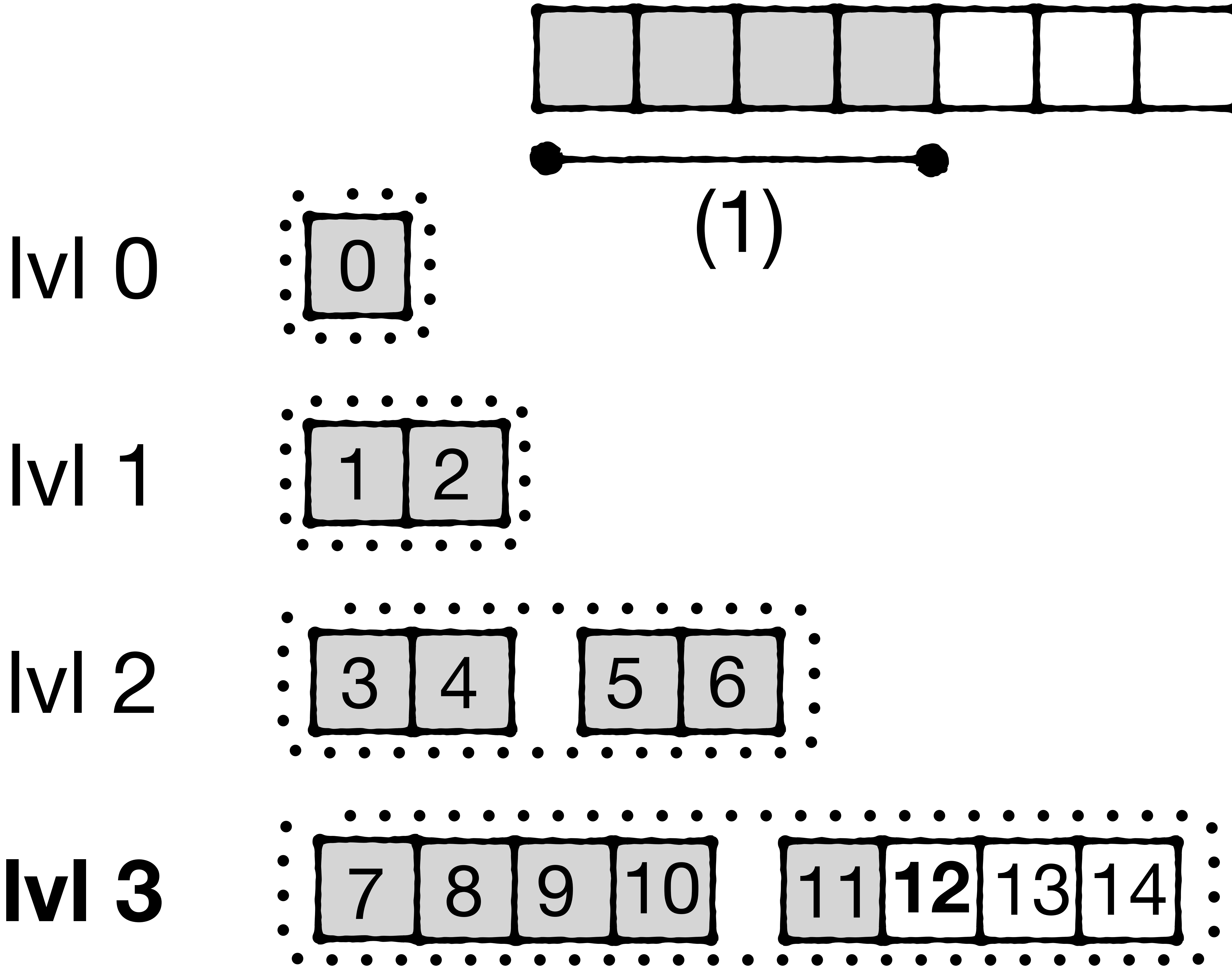
\# blocks in levels 0 to k-1

$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

$$= 2^{(\mathbf{k >> 1})} \cdot (2 + (k \,\&\, 1)) - 2$$

Shift by 1 bit to right

get(i)

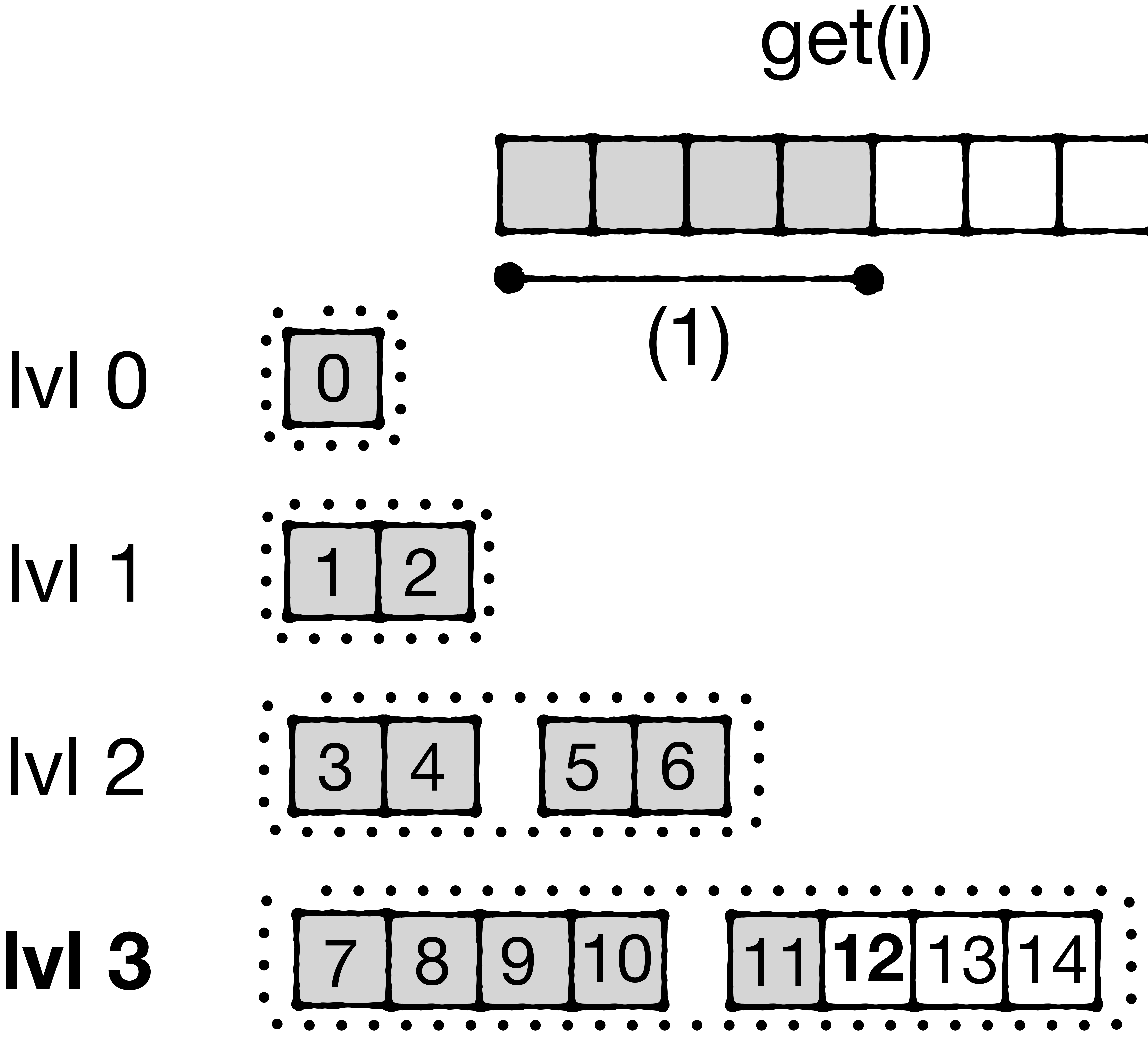


blocks in levels 0 to k-1

$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$

= (1 **<<** (k >> 1)) · (2 + (**k & 1**)) - 2

Shift to left

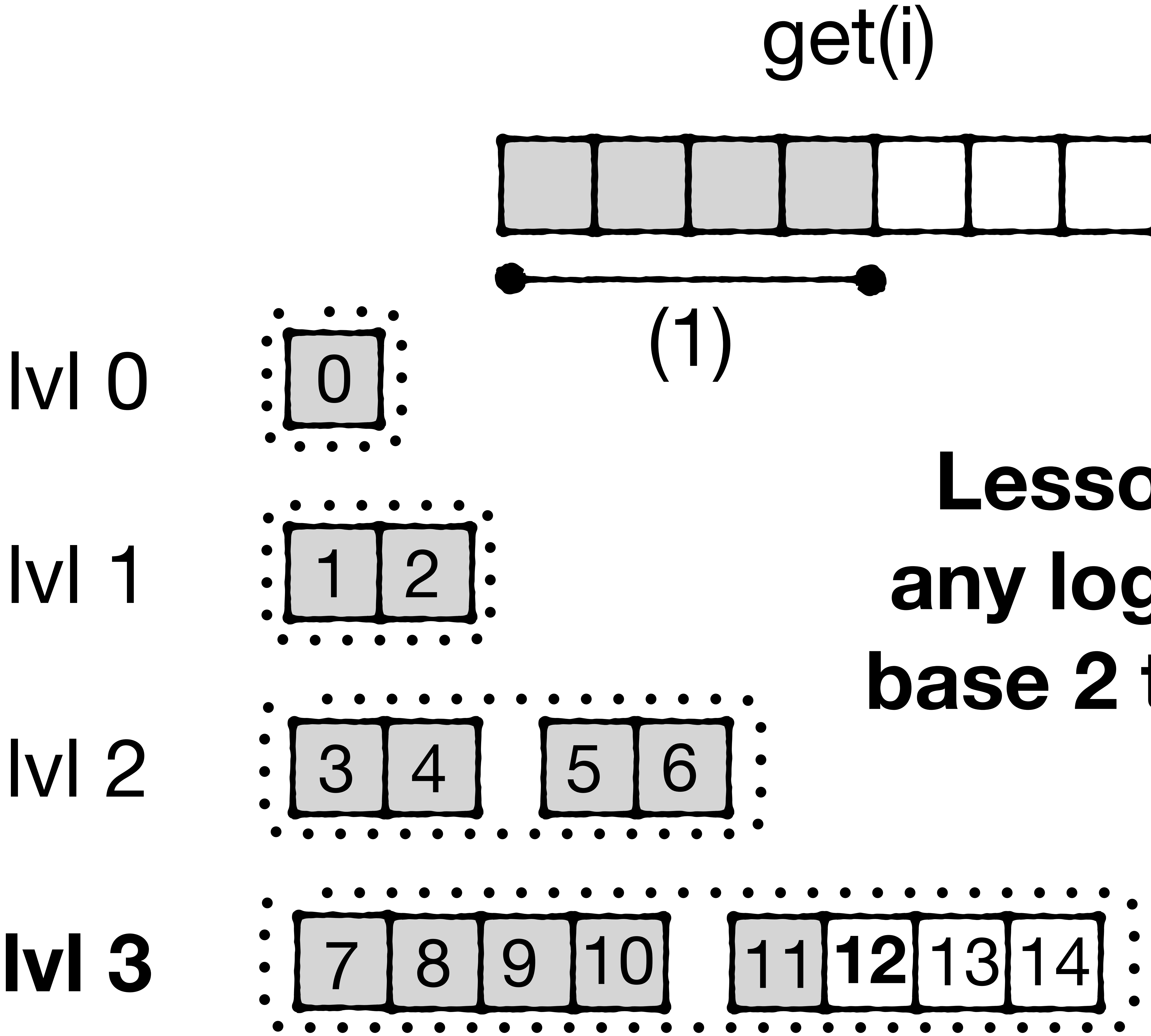


blocks in levels 0 to k-1

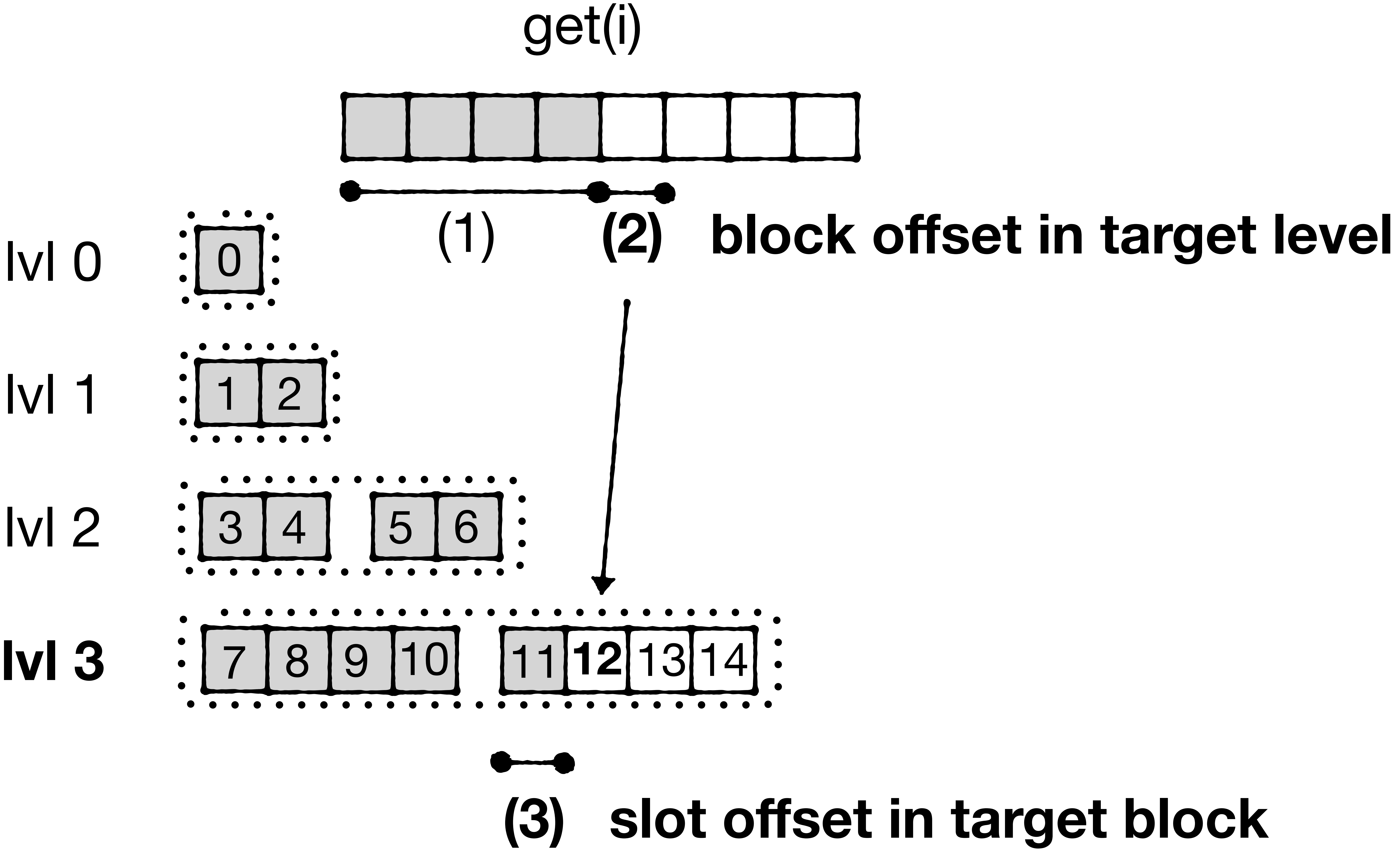
$$= 2^{\lfloor k/2 \rfloor} \cdot (2 + (k \bmod 2)) - 2$$

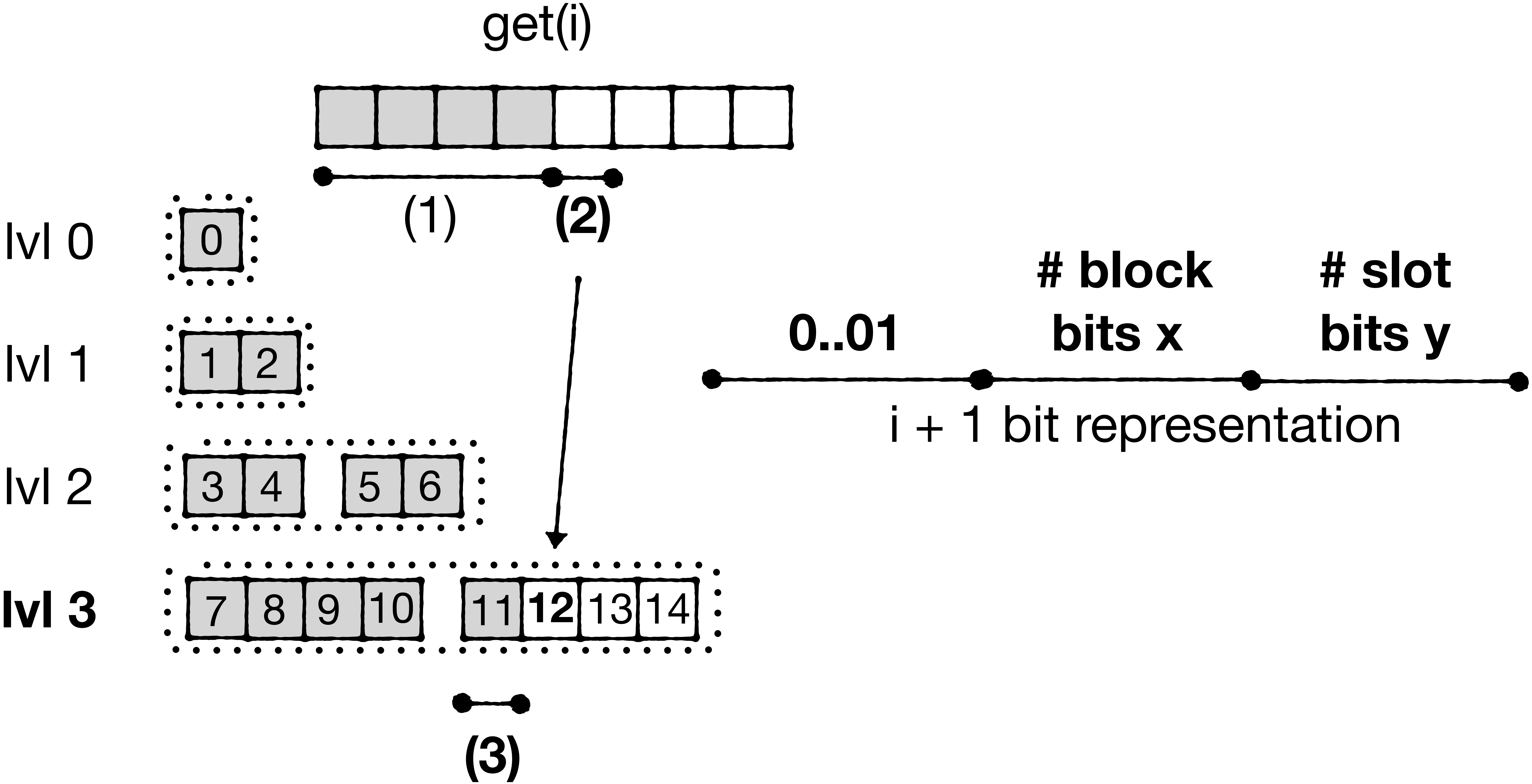
= (1 << (k >> 1)) · (2 + (k & 1)) - 2

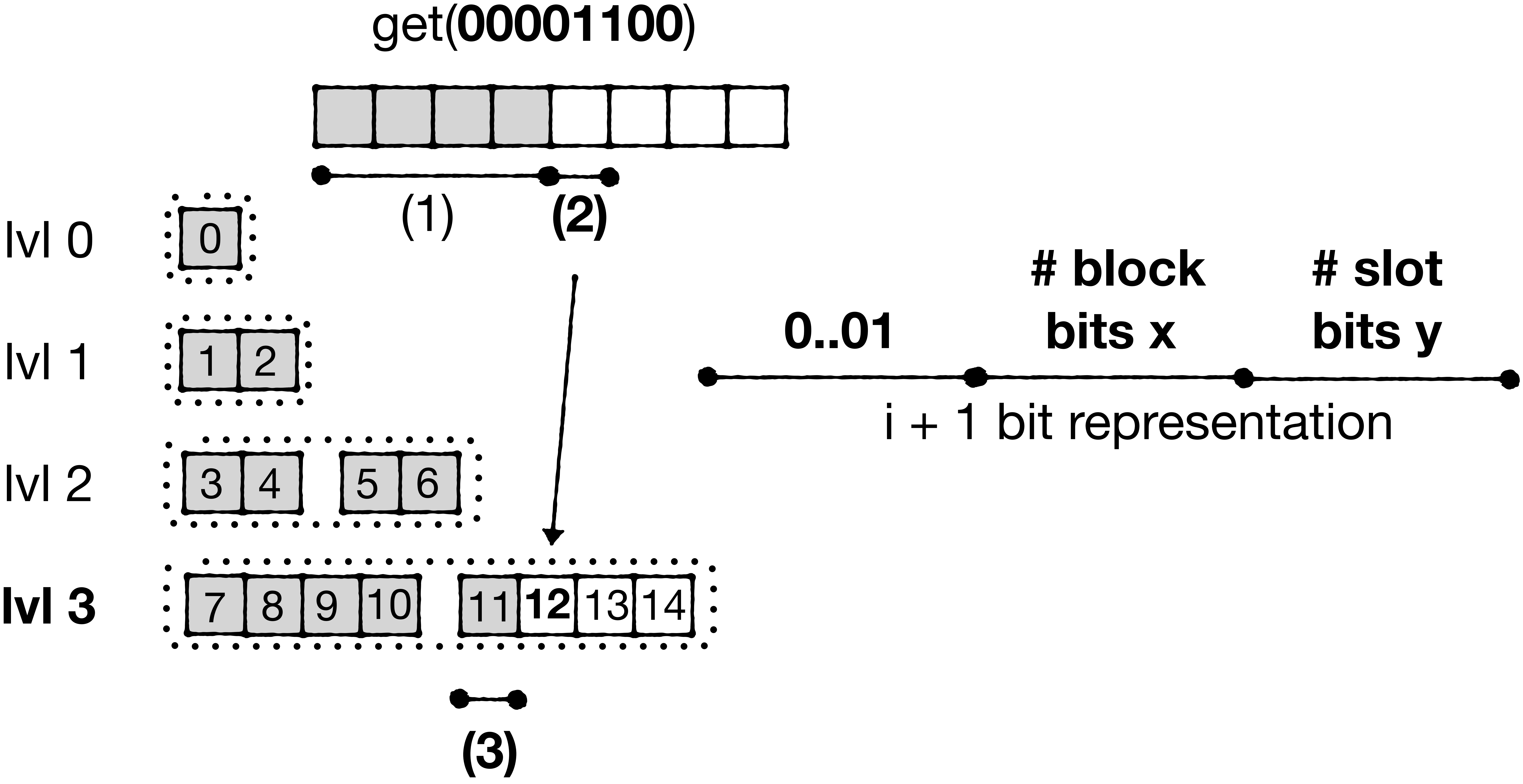
≈0.6 ns rather than ≈10ns

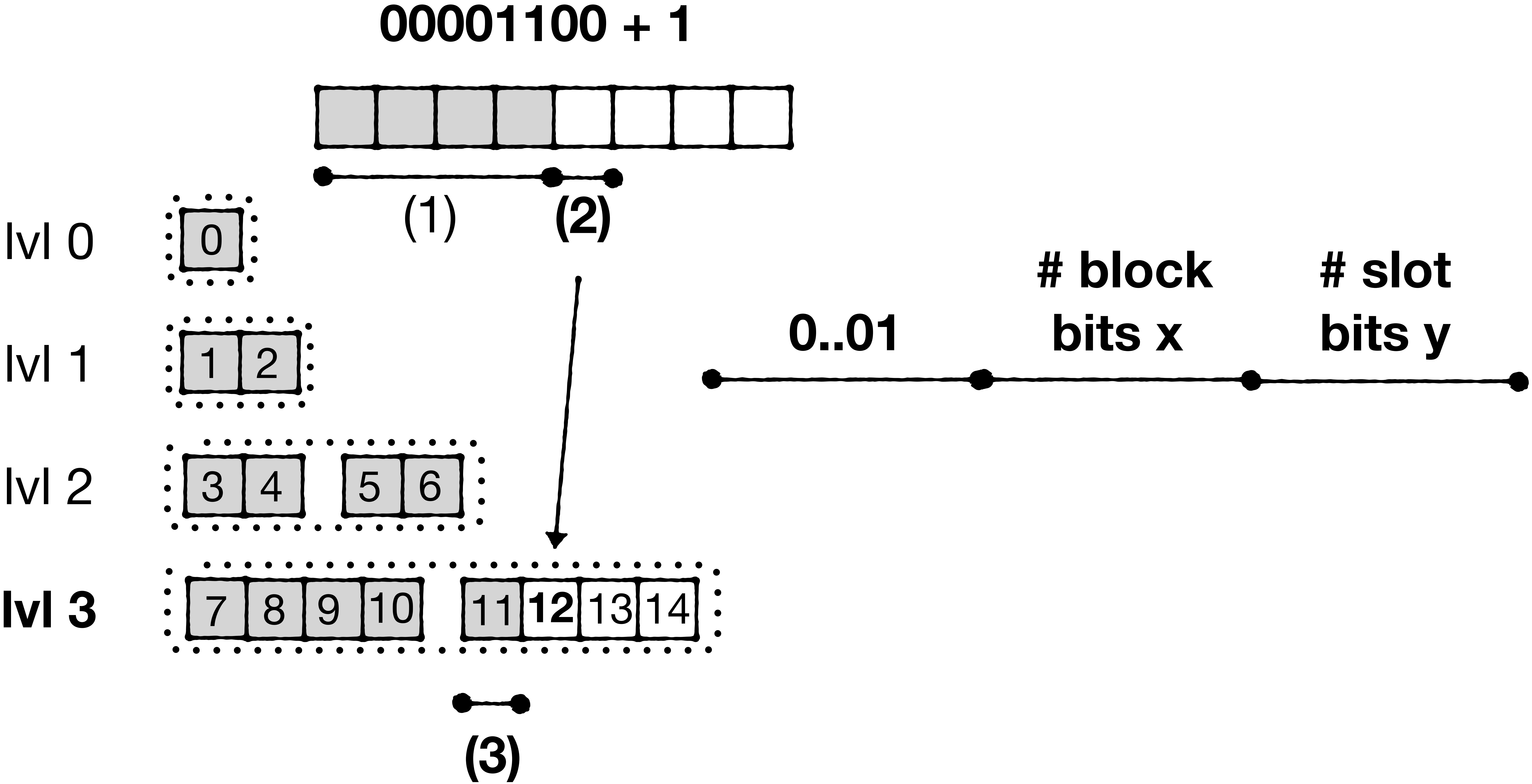


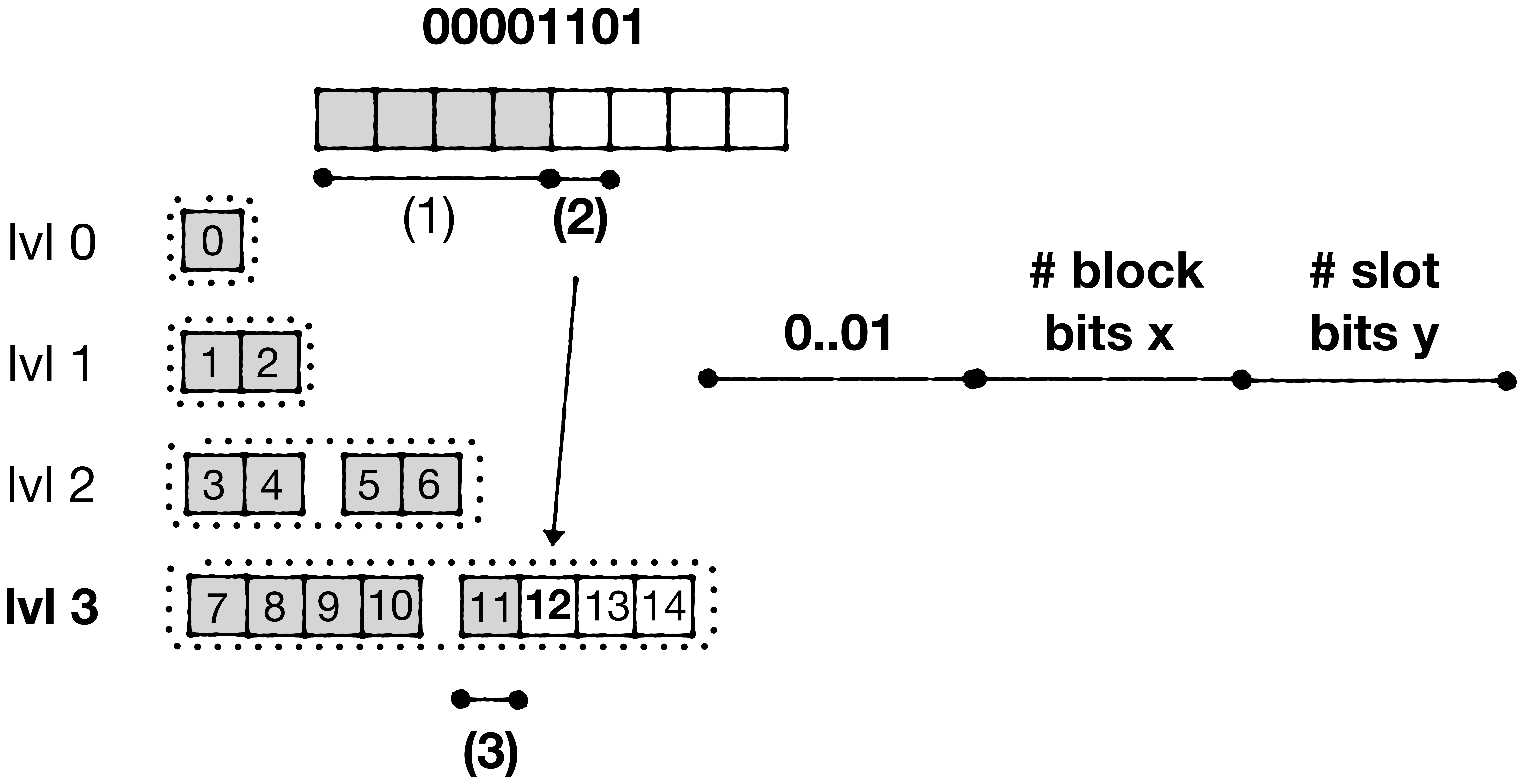
Lesson: design structure such that any log, division, or exponentiation is base 2 to support fast CPU operations

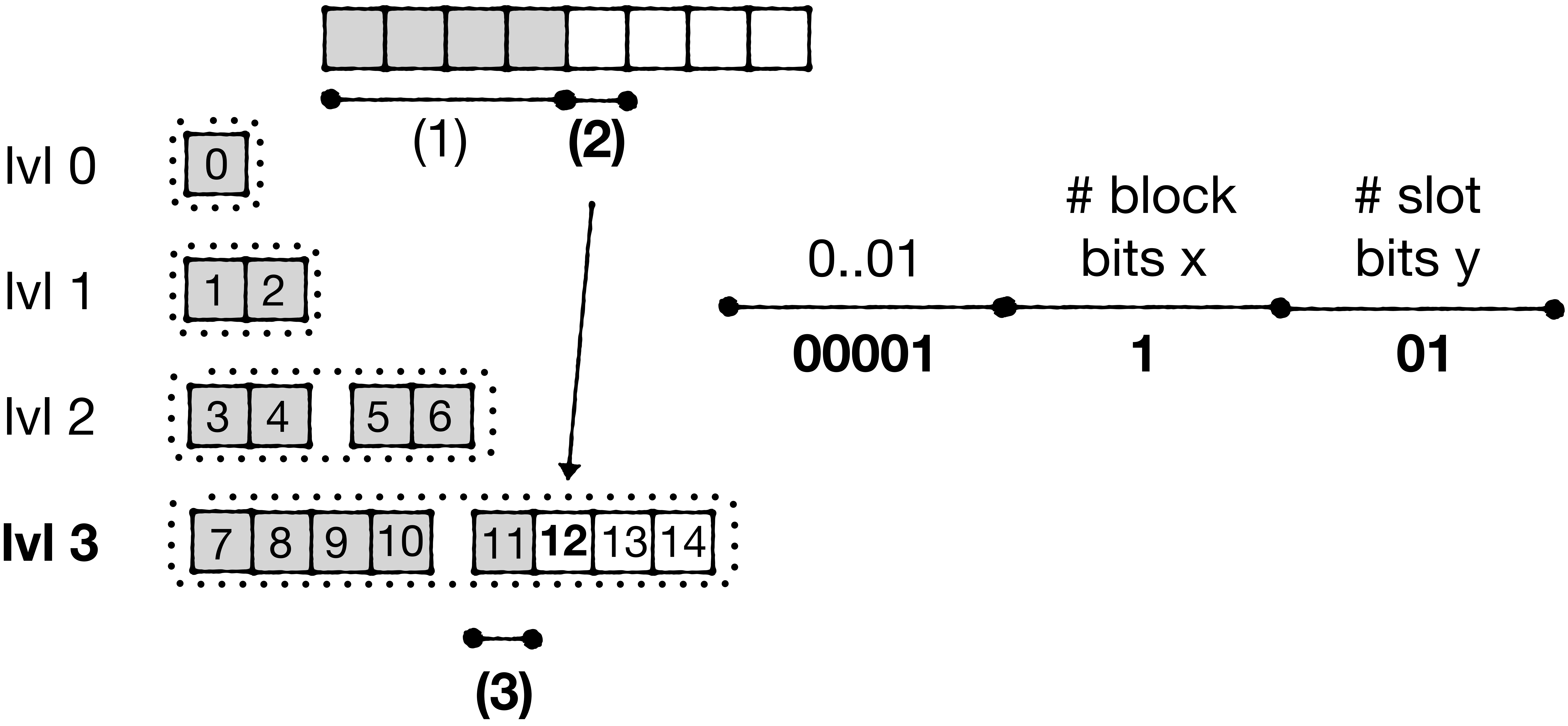

get(i)
(1)
(2) block offset in target level
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3) slot offset in target block


get(i)
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y
i + 1 bit representation

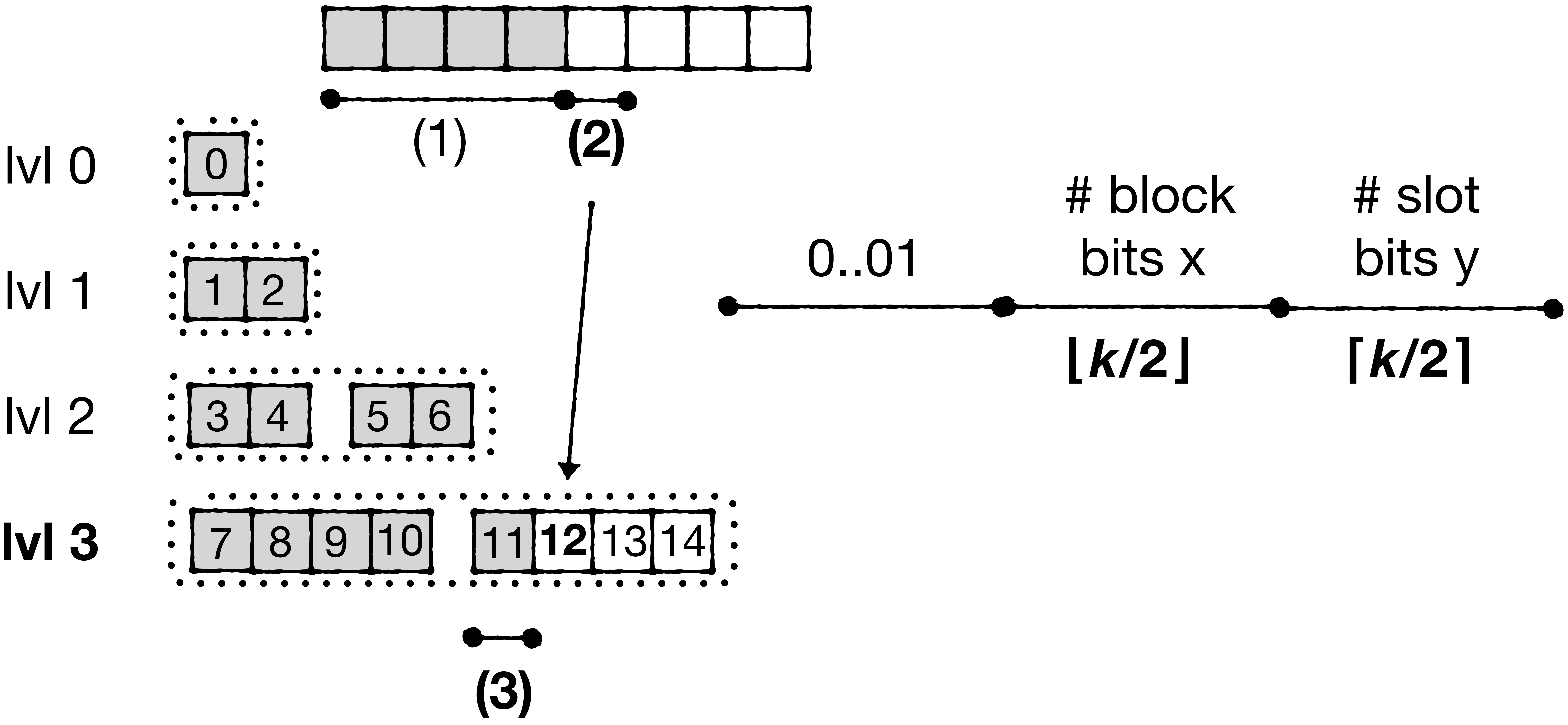

get(**00001100**)
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 **12** 13 14
(3)
 # block bits x
 # slot bits y
0..01
i + 1 bit representation


00001100 + 1
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y

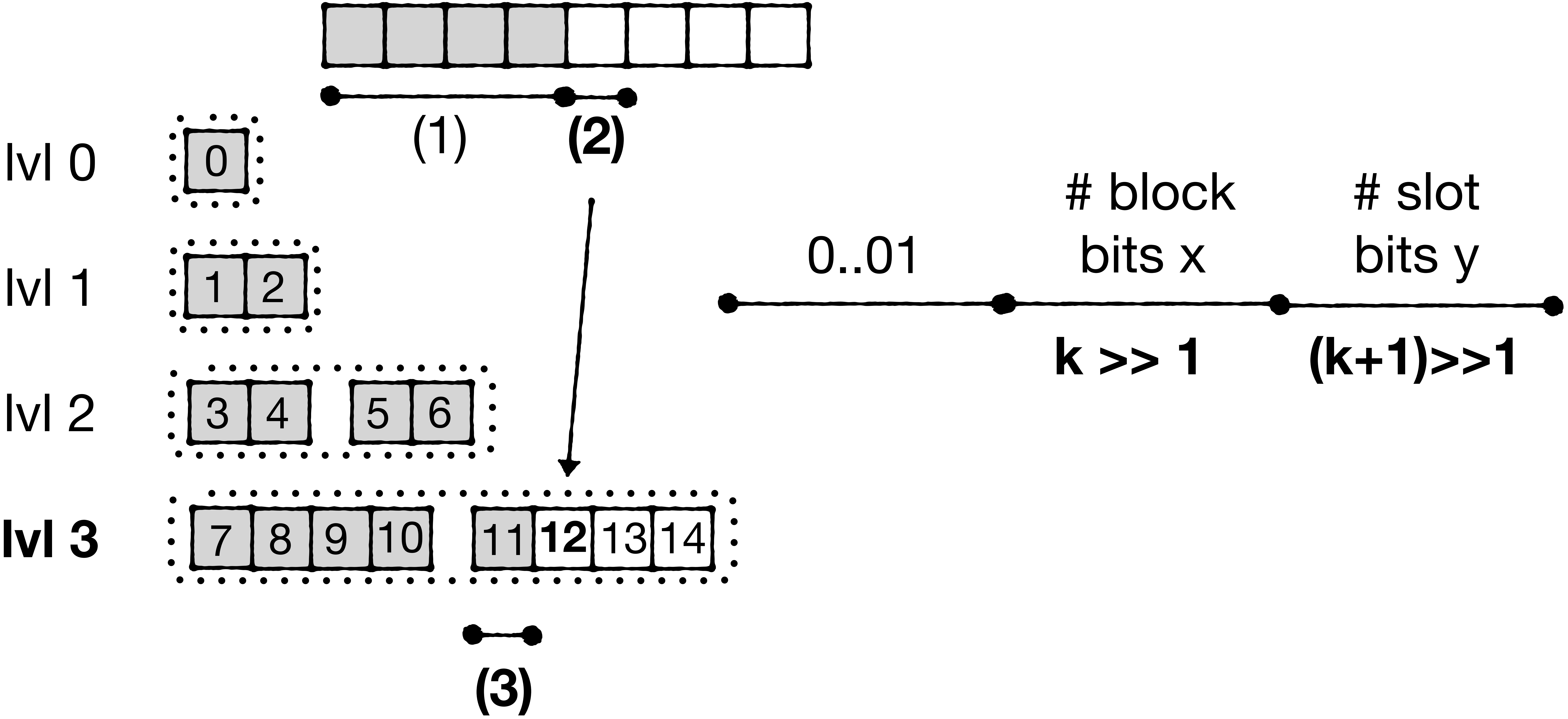

00001101
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y

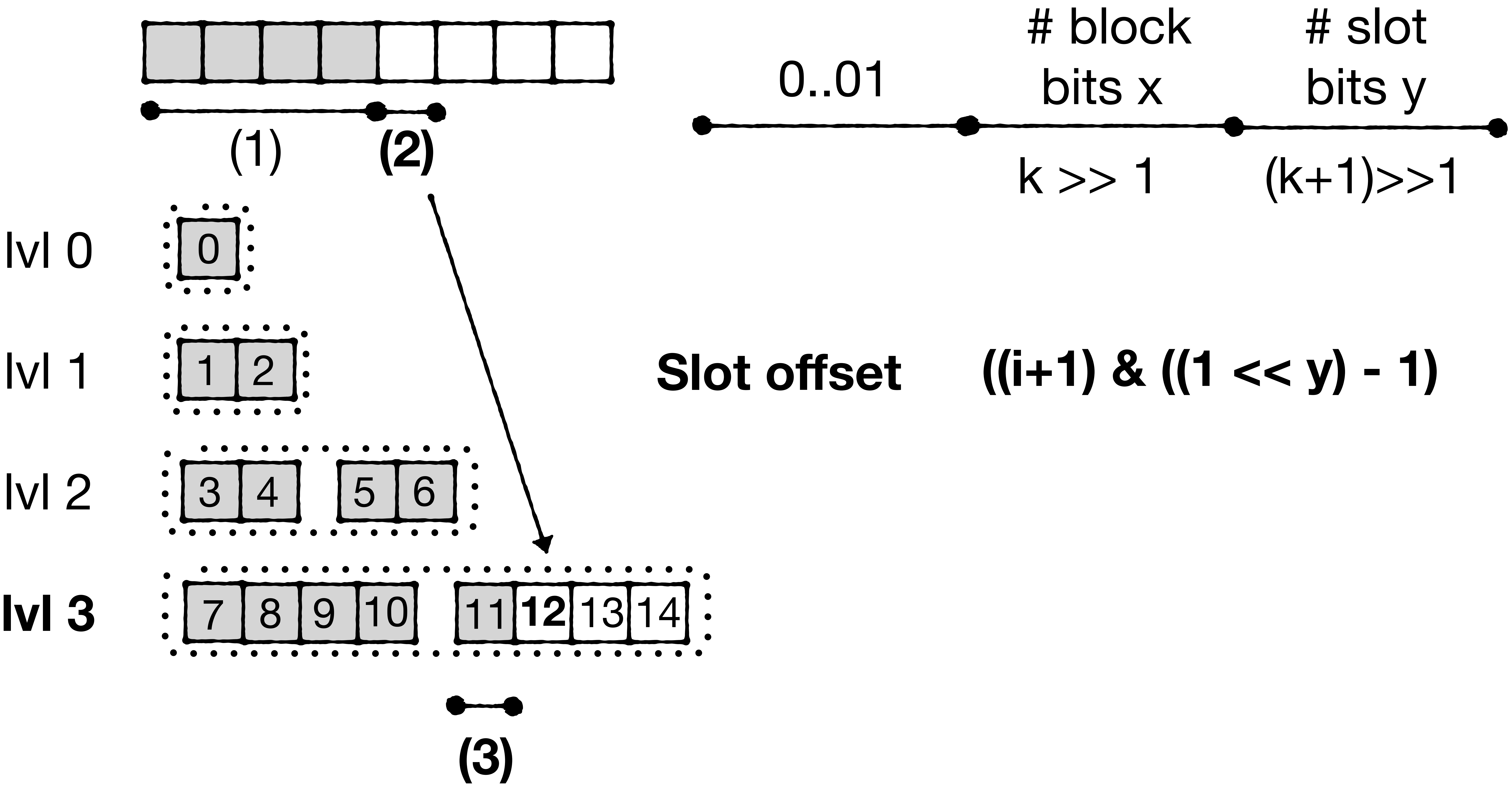


(1)
(2)
lvl 0
0
lvl 1
1
2
lvl 2
3
4
5
6
lvl 3
7
8
9
10
11
12
13
14
(3)
block
slot
0..01
bits x
bits y
00001
1
01

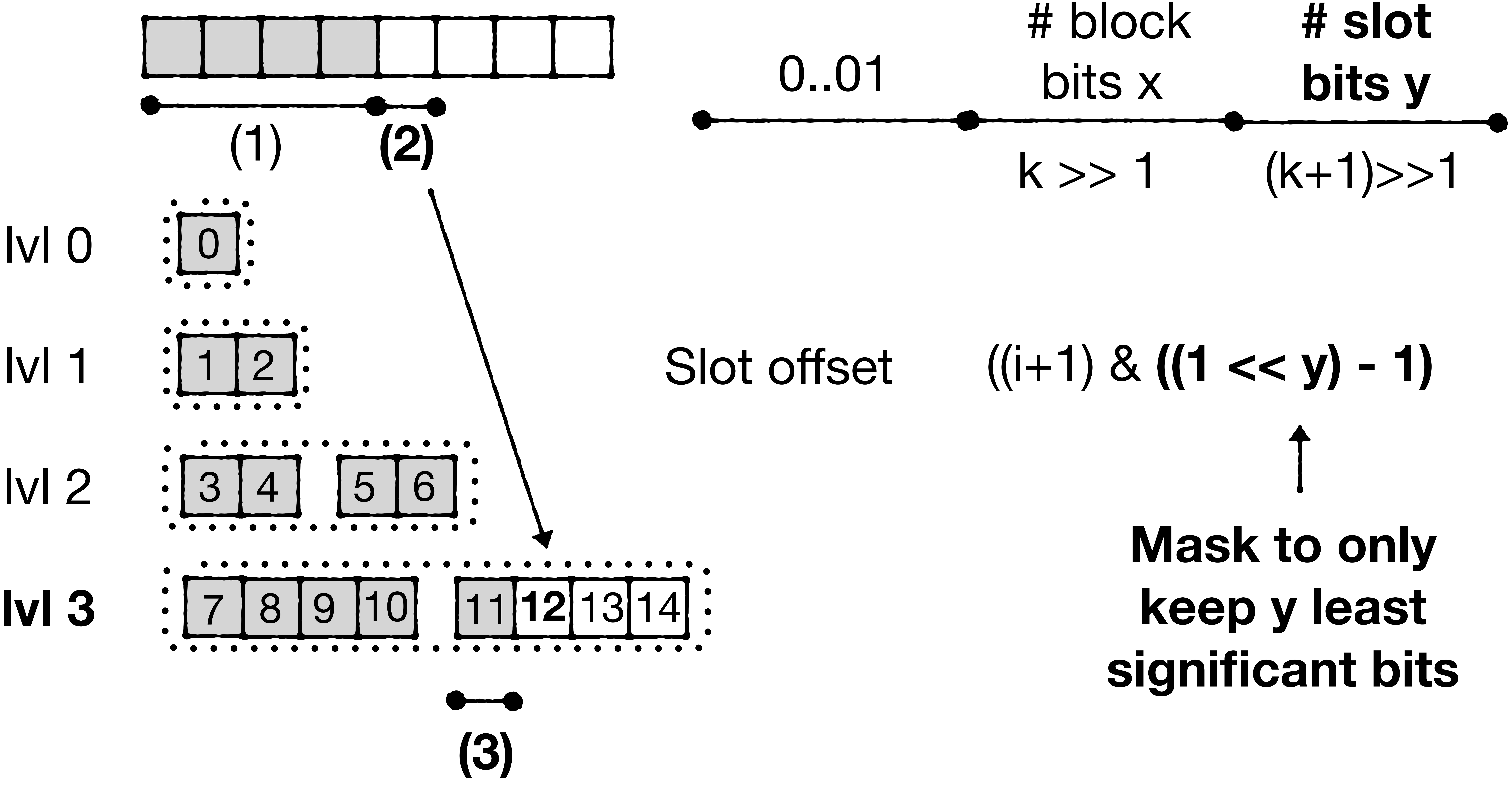


(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y
⌊k/2⌋
⌈k/2⌉

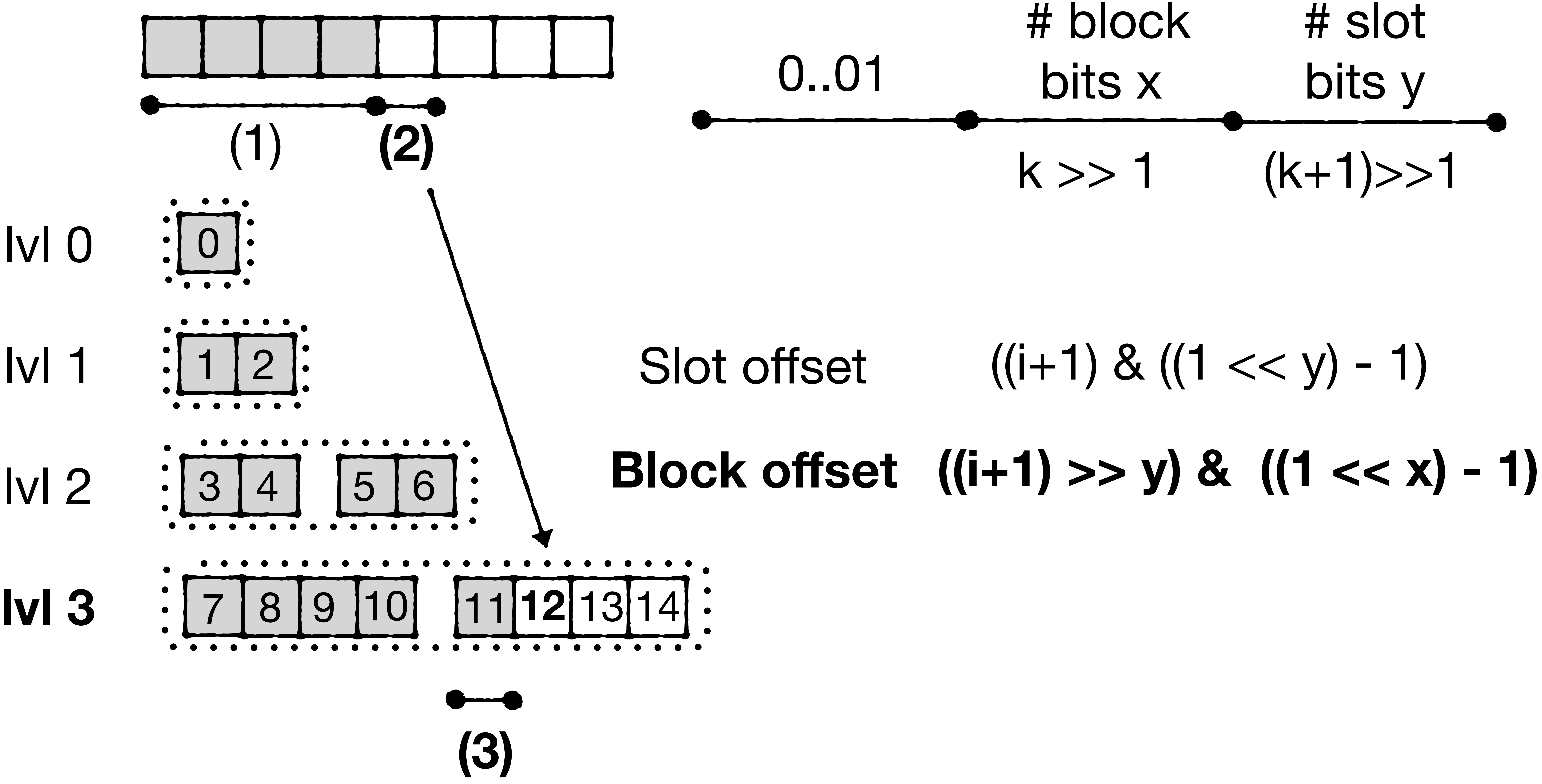

lvl 0
lvl 1
lvl 2
lvl 3
0
1 2
3 4 5 6
7 8 9 10 11 12 13 14
(1)
(2)
(3)
0..01
block bits x
slot bits y
k >> 1
(k+1)>>1



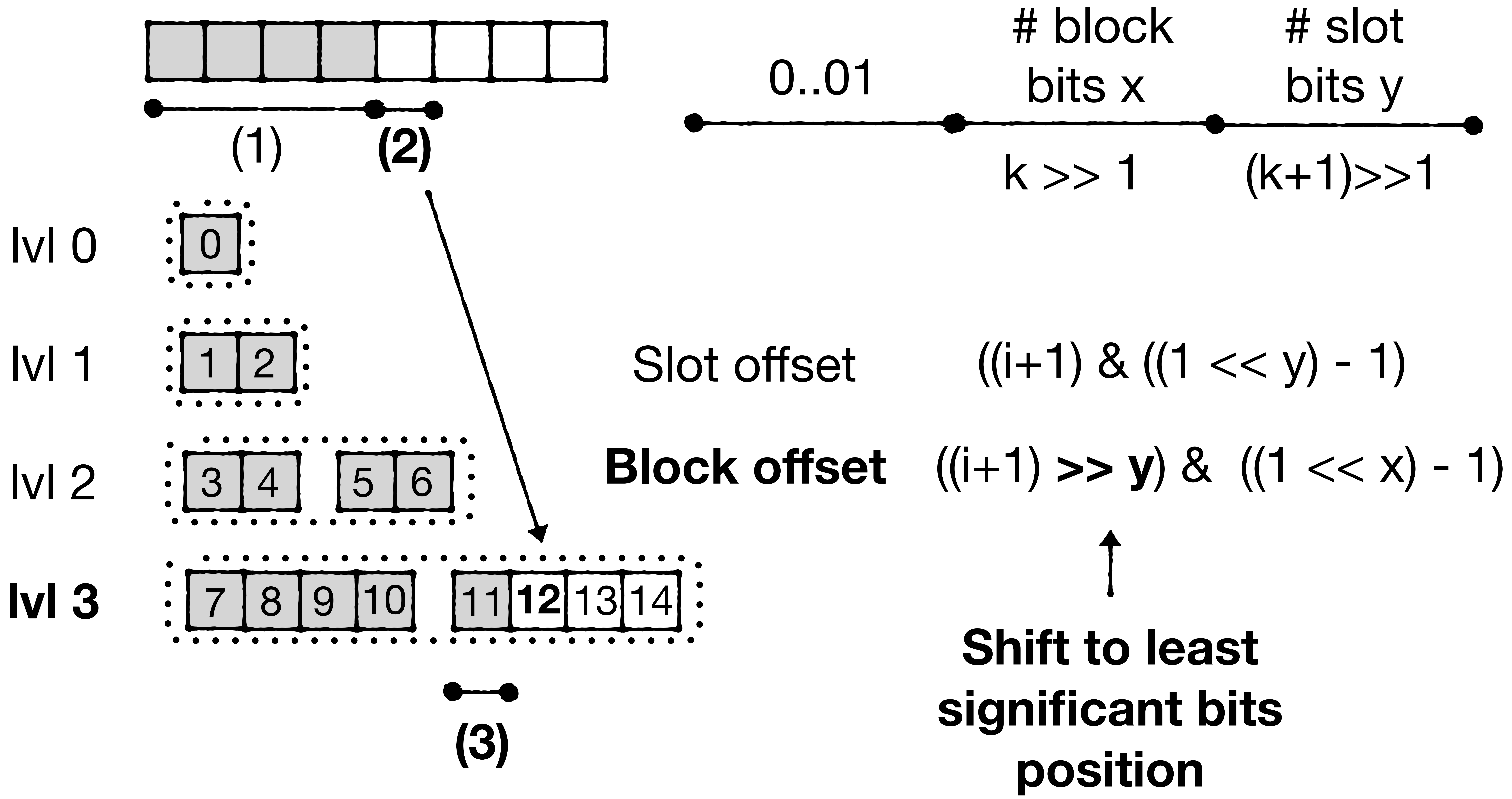
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y
k >> 1
(k+1)>>1
Slot offset
((i+1) & ((1 << y) - 1)



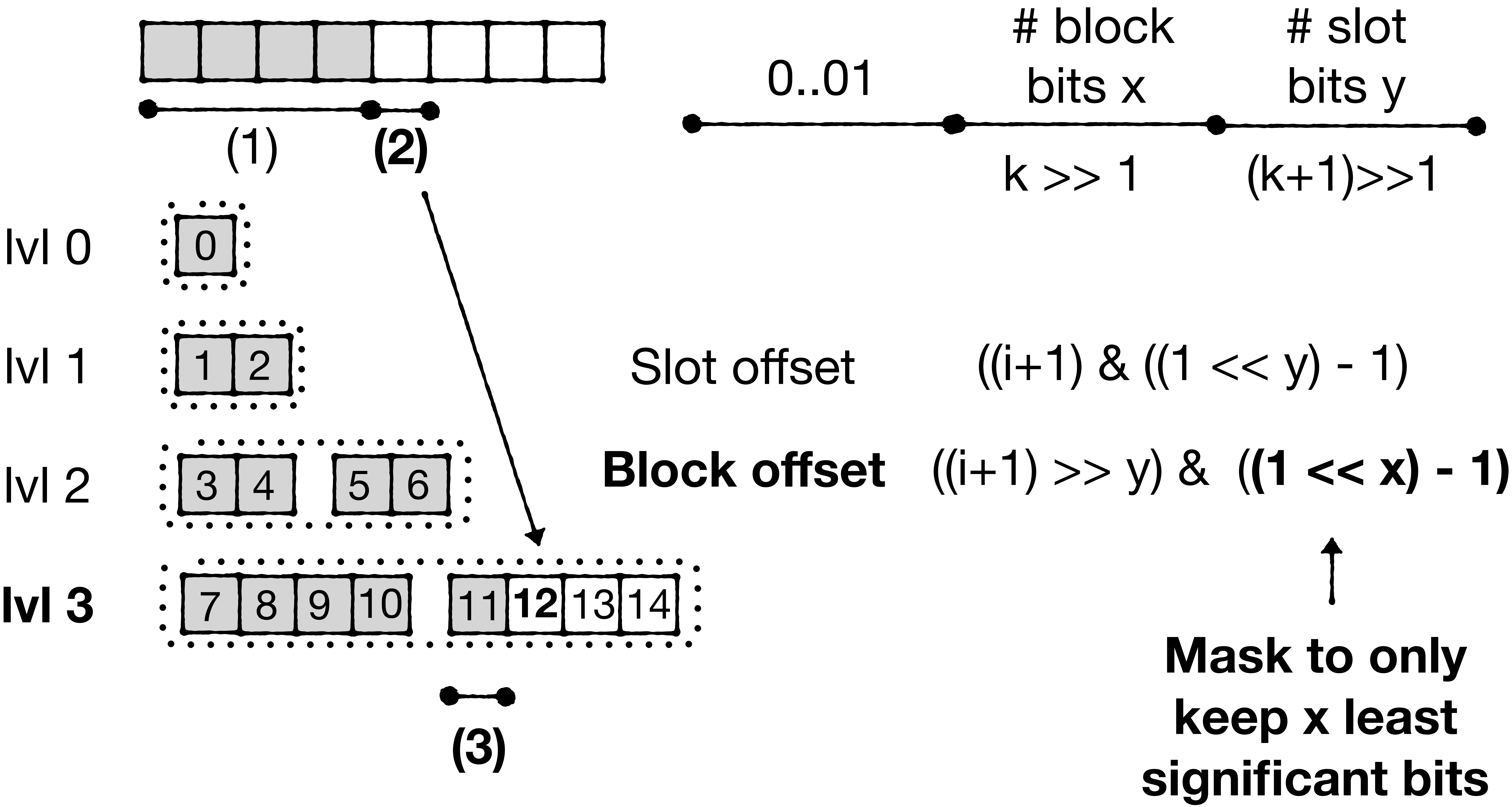
(1)
(2)
lvl 0
lvl 1
lvl 2
lvl 3
0
1 2
3 4 5 6
7 8 9 10 11 12 13 14
(3)
0..01
block bits x
slot bits y
k >> 1
(k+1)>>1
Slot offset
((i+1) & ((1 << y) - 1))
Mask to only keep y least significant bits



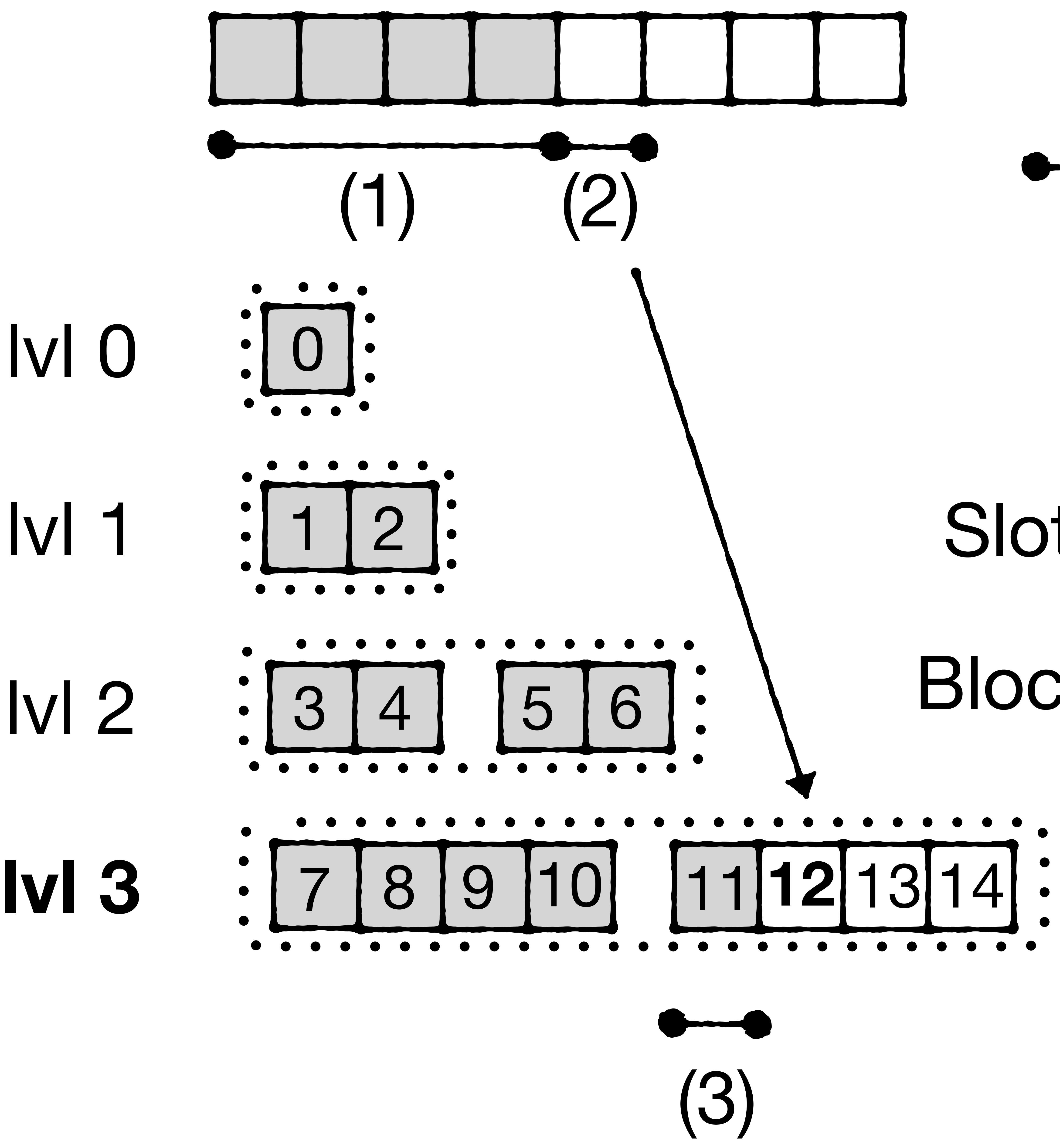
(1)
(2)
0..01
block bits x
slot bits y
k >> 1
(k+1)>>1
lvl 0
lvl 1
lvl 2
lvl 3
0
1 2
3 4 5 6
7 8 9 10 11 12 13 14
(3)
Slot offset ((i+1) & ((1 << y) - 1)
Block offset ((i+1) >> y) & ((1 << x) - 1)

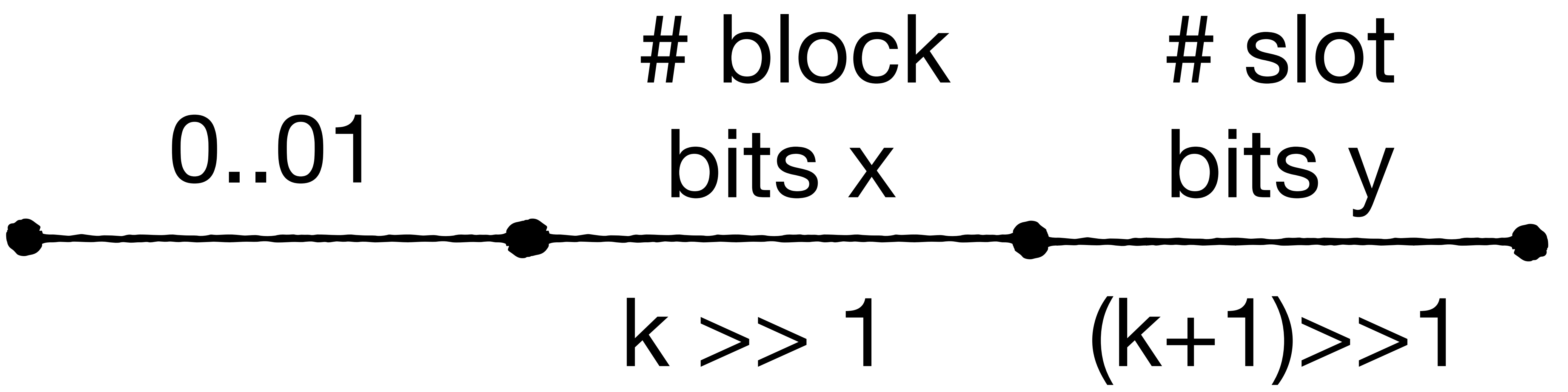


0..01
block bits x
slot bits y
k >> 1
(k+1)>>1
(1)
(2)
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
Slot offset ((i+1) & ((1 << y) - 1)
Block offset ((i+1) >> y) & ((1 << x) - 1)
Shift to least significant bits position



(1)
(2)
0..01
block
bits x
slot
bits y
k >> 1
(k+1)>>1
lvl 0
0
lvl 1
1 2
lvl 2
3 4 5 6
lvl 3
7 8 9 10 11 12 13 14
(3)
Slot offset ((i+1) & ((1 << y) - 1)
Block offset ((i+1) >> y) & ((1 << x) - 1)
Mask to only
keep x least
significant bits





Slot offset ((i+1) & ((1 << y) - 1)

Block offset ((i+1) >> y) & ((1 << x) - 1)

We're done :)

	Write-amp	Space-amp	Read-amp
indirection	$O(1+N^{-0.5})$	$O(1+N^{-0.5})$	$O(1)$

	Write-amp	Space-amp	Read-amp
indirection	$O(1+N^{-0.5})$	$O(1+N^{-0.5})$	$O(1)$
No indirection	$\frac{G}{G - 1}$	$O(G)$	1

	Write-amp	Space-amp	Read-amp
indirection	$O(1+N^{-0.5})$	$O(1+N^{-0.5})$	$O(1)$
No indirection $G > \phi$	$\frac{G}{G - 1}$	$\frac{G}{G - 1} + G$	1
No indirection $G < \phi$	$\frac{G}{G - 1}$	G to G+1	1

Thank you :)