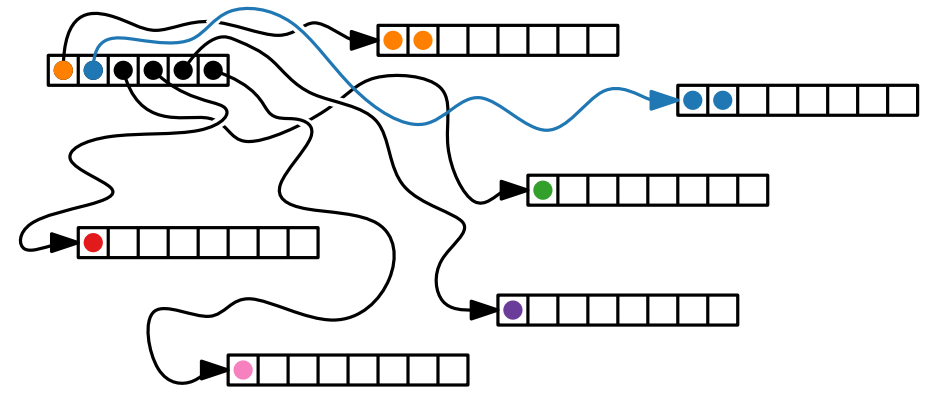
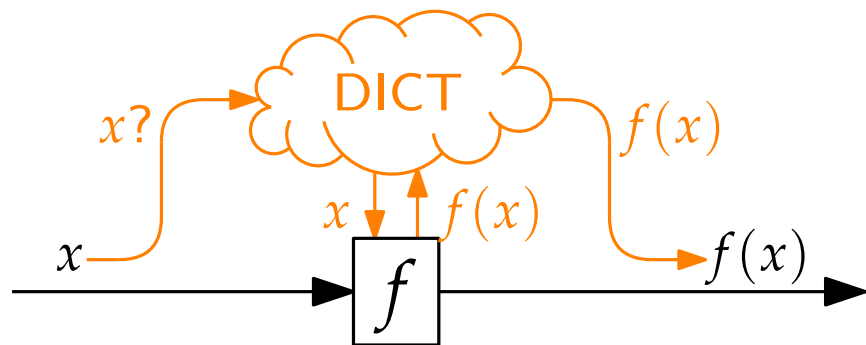


# Advanced Algorithms

## Algorithms in Practice

Computers are Fast · Knapsack DP · IntroSort · Swisstable

Tim Hegemann · WS23



# Theory & Practice

**Theorem.** Sorting an array of comparable objects (or primitives) on the Java Virtual Machine can be solved in  $\mathcal{O}(1)$  time.

Arrays on the JVM are indexed by 32-bit signed integers. So, an array cannot have more than  $2^{31} - 1$  entries.

**Observation.** In practice, the big  $O$  notation has some shortcomings...

**Remember:**

$$O(g) = \left\{ f: \mathbb{N} \rightarrow \mathbb{R} \left| \begin{array}{l} \text{there are positive constants } c \text{ and } n_0 \\ \text{such that for all } n \geq n_0: \\ f(n) \leq c \cdot g(n) \end{array} \right. \right\}$$

**What if?**

- We have lots of small instances but  $n_0$  is large?
- Two algorithms have similar  $g$  (e.g.  $n$  and  $n \log(n)$ ) but for one  $c$  is much higher?

*“In theory, there is no difference between  
theory and practice;  
but, in practice, there is.”*

# Do We Need More Detail?

[Knuth 1998]

	Step	Operations	Time
	$N^3$	CMPA, JG, JE	$3.5u$
Either	$N^4$	STA, INC	$3u$
	$N^5$	INC, LDA, CMPA, JGE	$6u$
Or	$N^8$	STX, INC	$3u$
	$N^9$	DEC, LDX, CMPX, JGE	$6u$

Thus about  $12.5u$  is spent on each record in each pass, and the total running time will be asymptotically  $12.5N \lg N$ , for both the average case and the worst case. This is slower than quicksort's average time, and it may not be enough better than heapsort to justify taking twice as much memory space, since the asymptotic running time of Program 5.2.3H is never more than  $18N \lg N$ .

## What about?

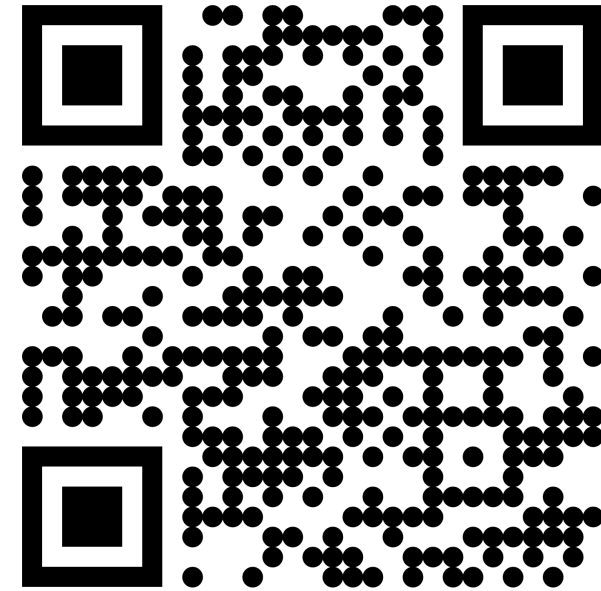
- Caching?
- Super-scalar Architectures?
- Out-of-order Processors?
- Vector Processors? Parallel Multiprocessors?
- Fixed-function Units?

# Computers are Fast

---

Event	Latency
1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	3 ns
Level 3 cache access	10 ns
RAM access	100 ns
SSD I/O	10–100 $\mu$ s
HDD I/O	1–10 ms
Internet (SF–NYC)	40 ms

---



<https://computers-are-fast.github.io/>

[Brendan Gregg. “Systems Performance: Enterprise and the Cloud.” 2nd Edition 2020]

# Advanced Algorithms

## Algorithms in Practice

### Implementing the Knapsack Dynamic Program

Tim Hegemann · WS23

# KNAPSACK

**Input:** Finite set  $U$ , for each  $u \in U$  a size  $s(u) \in \mathbb{Z}^+$  and a value  $v(u) \in \mathbb{Z}^+$ , and positive integers  $B$  and  $K$ .

**Task:** Is there a subset  $U' \subseteq U$  such that  $\sum_{u \in U'} s(u) \leq B$  and such that  $\sum_{u \in U'} v(u) \geq K$ ?

[Garey, Johnson 1979; Karp 1972]

**Optimization Problem:** Maximize  $K$

**Example:**

\$10	△	4
------	---	---

\$12	△	6
------	---	---

\$4	△	1
-----	---	---

\$6	△	2
-----	---	---

,  $B = 6$

**Strategies:**

	$\sum_{u \in U'} s(u)$	$\sum_{u \in U'} v(u)$
Greedy	6	12
Value/Size-Ratio	3	10
OPT	6	16

KNAPSACK is (weakly) NP-complete but can be solved in pseudo-polynomial time by dynamic programming.

# A Dynamic Program for KNAPSACK

Pick an arbitrary order on the items  $U$ .

$f(i, s) = \max$  value with size limit  $s$  using only items  $1 \dots i$ .

**Base case:**

$$f(1, s) = \begin{cases} v(1) & \text{if } s(1) \leq s \\ 0 & \text{otherwise} \end{cases}$$

**Recursion:**

$$f(i, s) = \max \begin{cases} f(i-1, s) \\ v(i) + f(i-1, s - s(i)) \end{cases} \quad \text{if } s \geq s(i)$$

**Result:**

$$f(|U|, B)$$

**Runtime:**

$$\mathcal{O}(|U| \cdot B) \longleftarrow \text{“pseudo-polynomial”}$$



# Implement it

```
case class Item(weight: Int, value: Int)

def solve(items: IndexedSeq[Item], weight: Int) =
  def go(i: Int, weight: Int): Int =
    val item = items(i)
    if i == 0 then
      if item.weight <= weight then item.value else 0
    else if item.weight <= weight then
      go(i - 1, weight) max
        (go(i - 1, weight - item.weight) + item.value)
    else go(i - 1, weight)

  go(items.size - 1, weight)
```

# Expectation Management

[Skiena 2012]

$n$	$f(n)$	$\lg n$	$n$	$n \lg n$	$n^2$	$2^n$	$n!$
10		0.003 $\mu s$	0.01 $\mu s$	0.033 $\mu s$	0.1 $\mu s$	1 $\mu s$	3.63 ms
20		0.004 $\mu s$	0.02 $\mu s$	0.086 $\mu s$	0.4 $\mu s$	1 ms	77.1 years
30		0.005 $\mu s$	0.03 $\mu s$	0.147 $\mu s$	0.9 $\mu s$	1 sec	$8.4 \times 10^{15}$ yrs
40		0.005 $\mu s$	0.04 $\mu s$	0.213 $\mu s$	1.6 $\mu s$	18.3 min	
50		0.006 $\mu s$	0.05 $\mu s$	0.282 $\mu s$	2.5 $\mu s$	13 days	
100		0.007 $\mu s$	0.1 $\mu s$	0.644 $\mu s$	10 $\mu s$	$4 \times 10^{13}$ yrs	
1,000		0.010 $\mu s$	1.00 $\mu s$	9.966 $\mu s$	1 ms		
10,000		0.013 $\mu s$	10 $\mu s$	130 $\mu s$	100 ms		
100,000		0.017 $\mu s$	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 $\mu s$	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 $\mu s$	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 $\mu s$	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 $\mu s$	1 sec	29.90 sec	31.7 years		

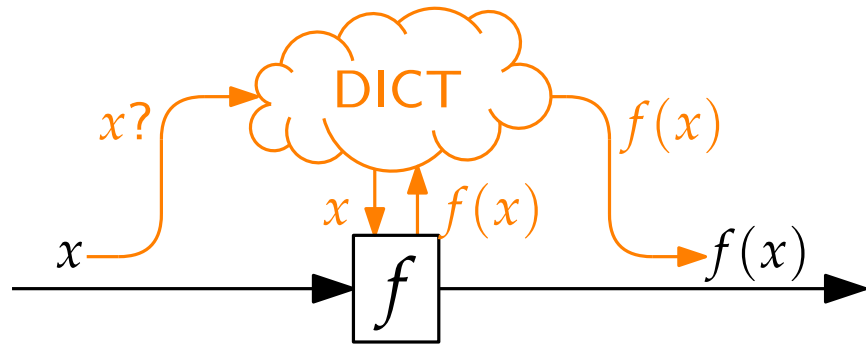
Figure 2.4: Growth rates of common functions measured in nanoseconds

# Running Times

$|U| =$       30      100  
Scala naïve   1.15 s    $\approx 2700 \times$  age of the universe

$$B = 10|U|$$

# Memoization



```
def memoized[K, V](f: K => V): K => V =  
  val dict = mutable.Map.empty[K, V]  
  (key: K) => dict.getOrElseUpdate(key, f(key))
```

```
lazy val go: ((Int, Int)) => Int =  
  memoized((args: (Int, Int)) =>  
    val (i, weight) = args  
    // fill in the body of the old go  
  )
```

# Running Times

	$ U  =$	30	100	1K		$B = 10 U $
Scala naïve		1.15 s				
Scala memoized		230 ms	267 ms	...		

# Running Times

```
Exception in thread "main" java.lang.StackOverflowError
  at scala.runtime.BoxesRunTime.equalsNumNum(BoxesRunTime.java:148)
  at scala.runtime.BoxesRunTime.equalsNumObject(BoxesRunTime.java:138)
  at scala.runtime.BoxesRunTime.equals2(BoxesRunTime.java:127)
  at scala.runtime.BoxesRunTime.equals(BoxesRunTime.java:119)
  at scala.Tuple2.equals(Tuple2.scala:24)
  at scala.runtime.BoxesRunTime.equals2(BoxesRunTime.java:133)
  at scala.runtime.BoxesRunTime.equals(BoxesRunTime.java:119)
  at scala.collection.mutable.HashMap$Node.findNode(HashMap.scala:621)
  at scala.collection.mutable.HashMap.getOrElseUpdate(HashMap.scala:449)
  at y.Memoized$.memoized$$anonfun$1(Y.scala:64)
  at y.Memoized$.go$lzyINIT1$1$$anonfun$1(Y.scala:72)
  at y.Memoized$.memoized$$anonfun$1$$anonfun$1(Y.scala:64)
  at scala.collection.mutable.HashMap.getOrElseUpdate(HashMap.scala:454)
  at y.Memoized$.memoized$$anonfun$1(Y.scala:64)
  ...
```

# Running Times

$ U  =$	30	100	1K	10K	$B = 10 U $
Scala naïve	1.15 s				
Scala memoized	230 ms	267 ms	⚡ so		
Scala TCO		273 ms	2.30 s	...	

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at java.base/java.lang.Integer.valueOf(Integer.java:1077)
  at scala.runtime.BoxesRunTime.boxToInteger(BoxesRunTime.java:60)
  at y.TCO$.go$3(Y.scala:112)
  at y.TCO$.solve(Y.scala:121)
  at y.TCO$.runTco(Y.scala:82)
  at y.runTco.main(Y.scala:80)
```

Covered in this talk:



# Reducing the Memory Footprint

- Idea:**
- Use Iteration instead of Recursion
  - Use Arrays instead of a Dictionary

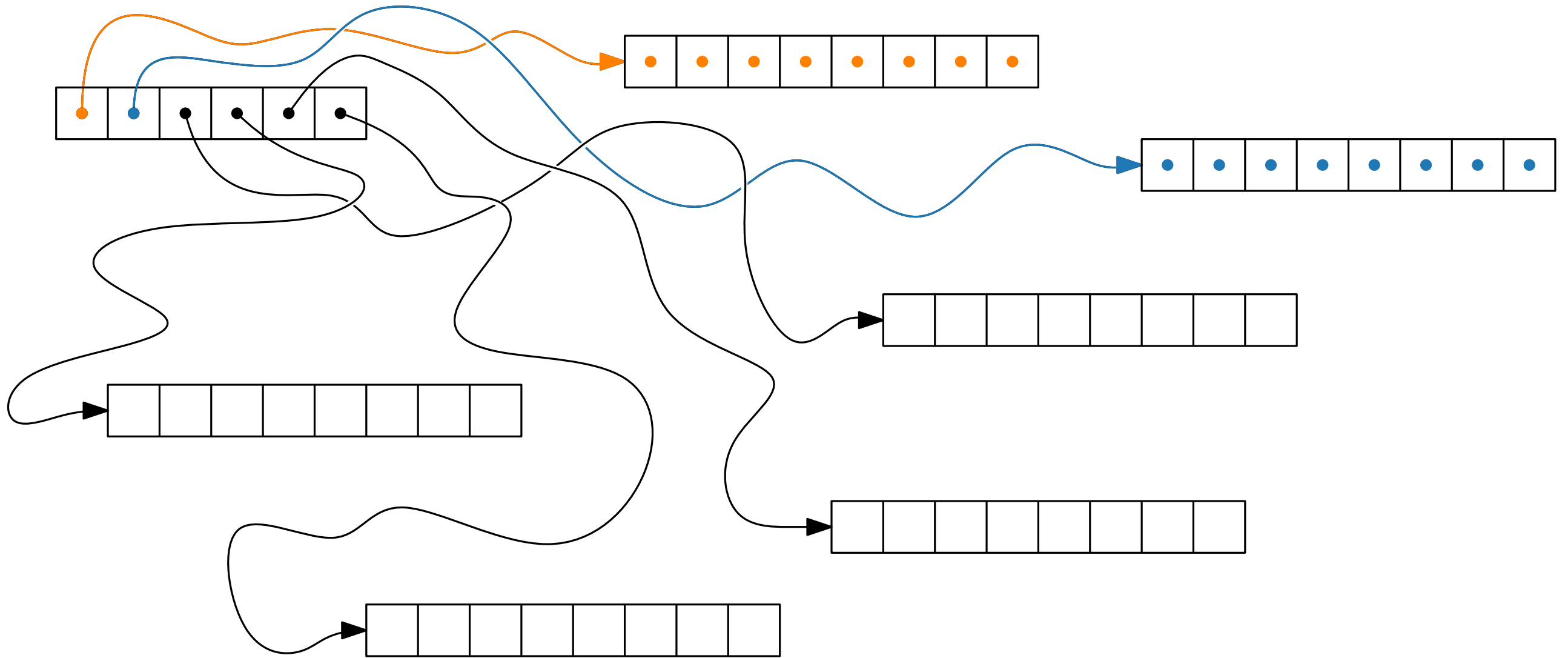


# Reducing the Memory Footprint

```
def solve(items: Seq[Item], weight: Int) =  
  val dict = Array.fill(items.size, weight + 1)(0)  
  val head = items.head  
  for w <- head.weight to weight do dict(0)(w) = head.value  
  for  
    (item, i) <- items.zipWithIndex.tail  
    w <- 0 to weight  
  do  
    val dontTake = dict(i - 1)(w)  
    if w >= item.weight then  
      val take = dict(i - 1)(w - item.weight) + item.value  
      dict(i)(w) = dontTake max take  
    else dict(i)(w) = dontTake  
  dict(items.size - 1)(weight)
```

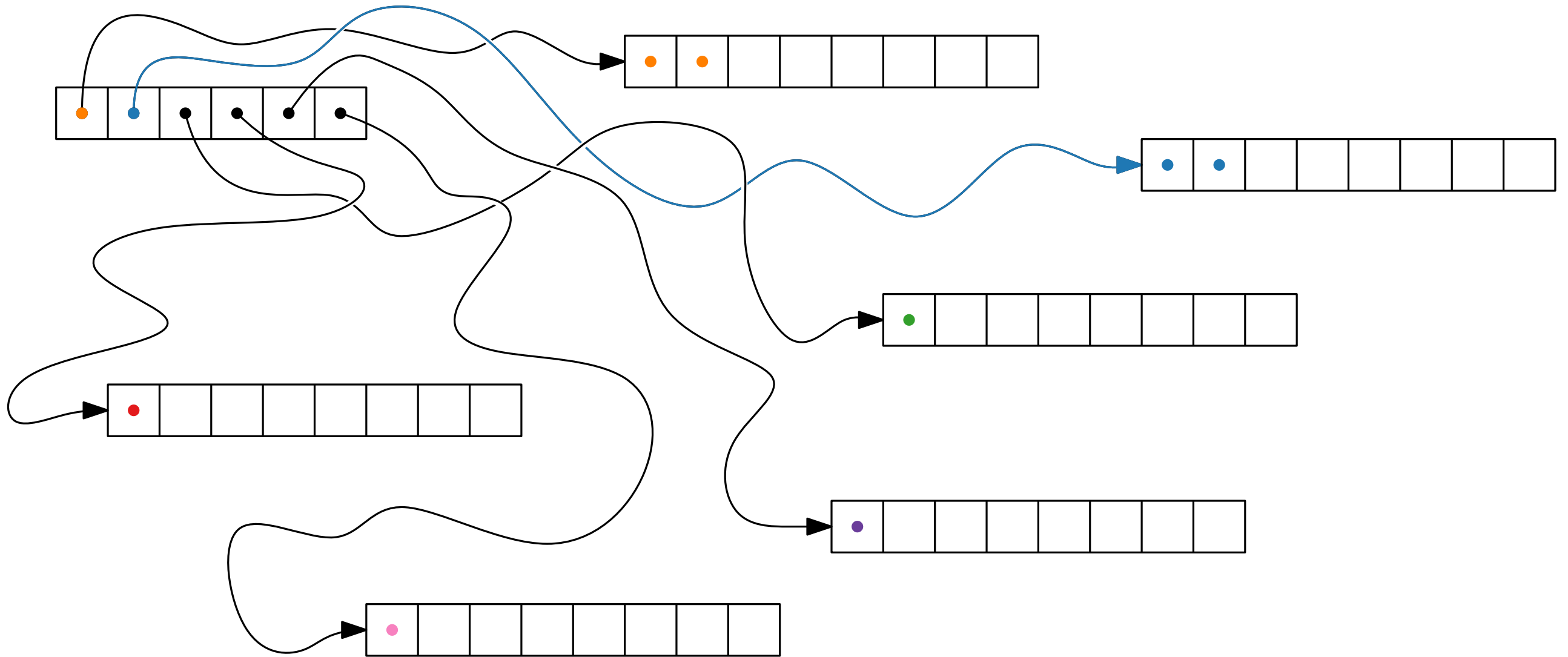
# Arrays and Memory Locality

`table(outer)(inner)`



# Arrays and Memory Locality

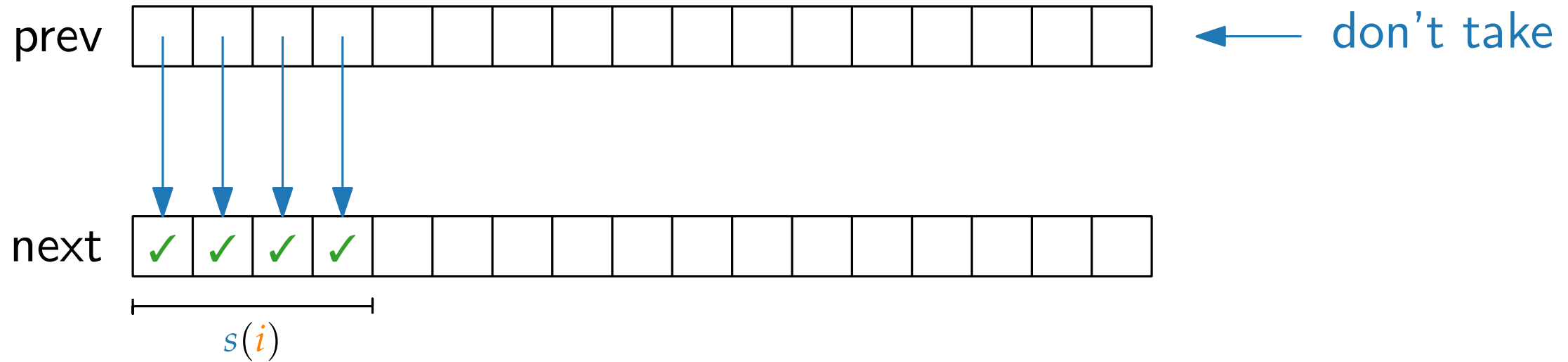
`table(inner)(outer)`



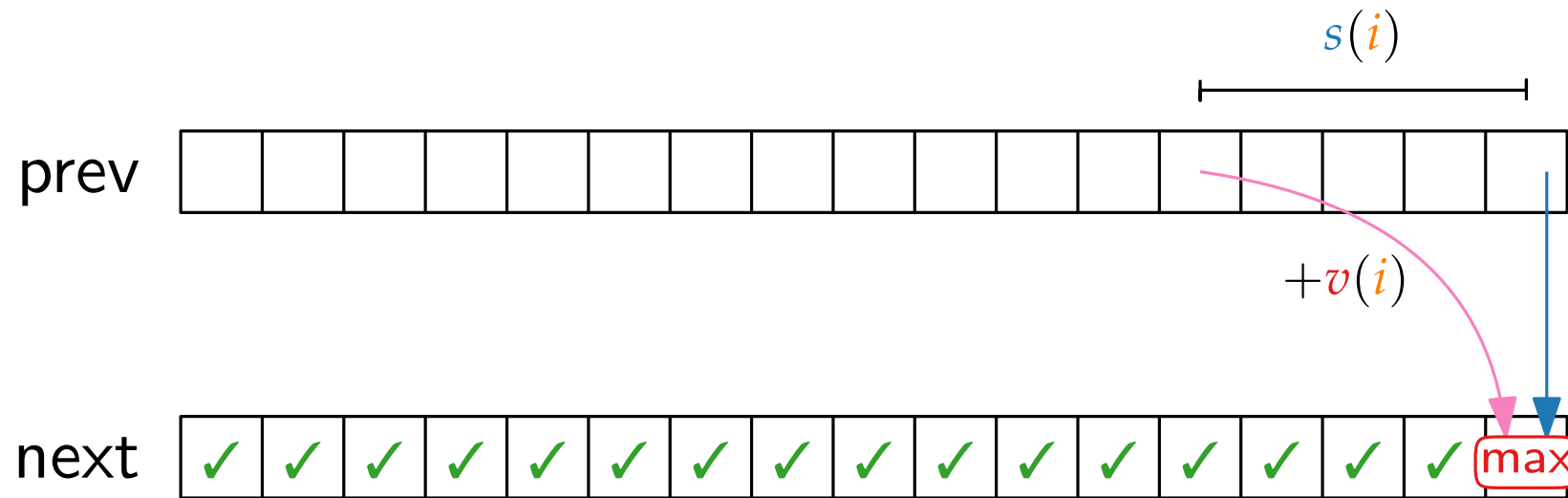
# Running Times

	$ U  =$	30	100	1K	10K		$B = 10 U $
Scala naïve		1.15 s					
Scala memoized		230 ms	267 ms	⚡ so			
Scala TCO			273 ms	2.30 s	⚡ oom		
Scala Arrays				458 ms	27.5 s		

$$f(i, s) = \max \begin{cases} f(i-1, s) & \leftarrow \text{don't take} \\ v(i) + f(i-1, s - s(i)) & \text{if } s \geq s(i) \end{cases}$$

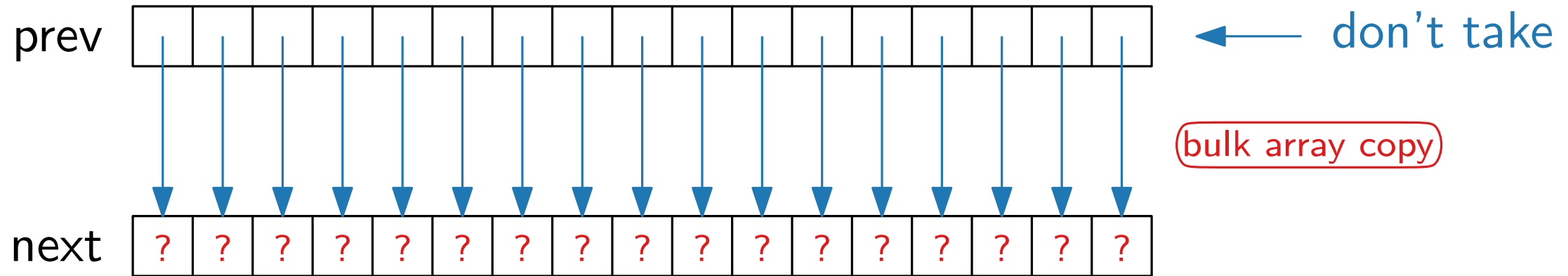


$$f(i, s) = \max \begin{cases} f(i-1, s) \\ v(i) + f(i-1, s - s(i)) & \text{if } s \geq s(i) \end{cases}$$



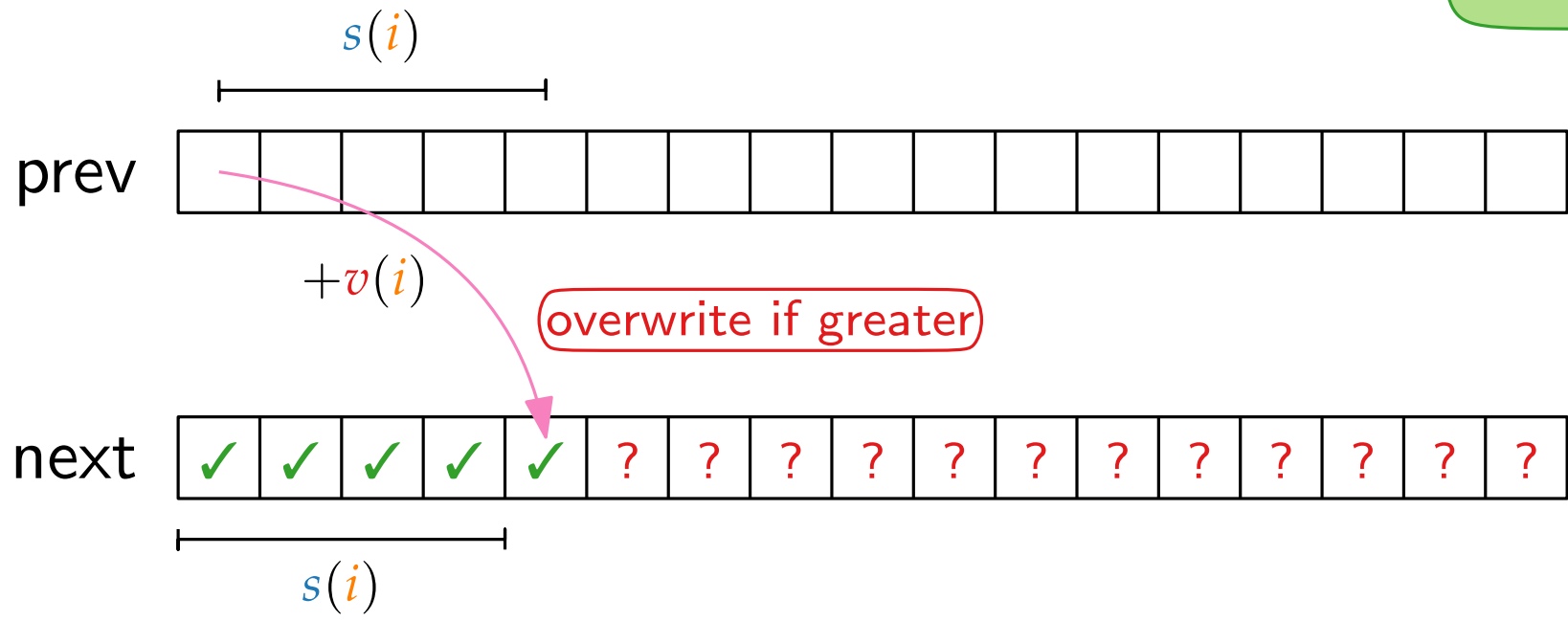
$$f(i, s) = \max \begin{cases} f(i-1, s) \\ v(i) + f(i-1, s - s(i)) \end{cases} \text{ if}$$

Hint: Use optimized procedures provided by your operating system!



$$f(i, s) = \max \begin{cases} f(i-1, s) \\ v(i) + f(i-1, s - s(i)) \end{cases} \text{ if}$$

Hint: Use optimized procedures provided by your operating system!





# Running Times

	$ U  = 30$	100	1K	10K	$B = 10 U $
Scala naïve	1.15 s				
Scala memoized	230 ms	267 ms	⚡ so		
Scala TCO		273 ms	2.30 s	⚡ oom	
Scala Arrays			458 ms	27.5 s	
Scala bulk copy				437 ms	
Java				636 ms	
node.js				957 ms	
Python3				96.8 s	
Pypy3				2.48 s	
Rust (safe)				300 ms	
C++ (by tvd)				212 ms	
Rust with AVX2				43 ms	

# Advanced Algorithms

## Algorithms in Practice

### How to Sort a Million 32-bit Integers

Tim Hegemann · WS23

# How to Sort a Million 32-bit Integers?

Well-known comparison-based sorting algorithms

	best-case	average	worst-case	in-place
Insertion sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	✓
Heap sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	✓
Quicksort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$ !	✓
<del>Merge sort</del>	<del><math>\mathcal{O}(n \log(n))</math></del>	<del><math>\mathcal{O}(n \log(n))</math></del>	<del><math>\mathcal{O}(n \log(n))</math></del>	<del>✗</del>

← very low constant factors

- Observation.**
- Equal integers are undistinguishable.
  - Integer comparisons are probably much cheaper than copying.

**What about sorting in linear time?**

~~Counting sort:  $\mathcal{O}(n + k)$~~

Radix sort:  $\mathcal{O}(w \cdot n)$   $w = 32$

# Quicksort With an Emergency Exit

**Observation.** Quicksort has optimal runtime as long as the recursion depth is  $\mathcal{O}(\log(n))$ .

**Idea.** If for any partition the recursion reaches a depth of  $a \log(n)$  (for a given parameter  $a$ ) stop and sort this section with heap sort.

$a = 2$

INTROSORT [Musser 1997]

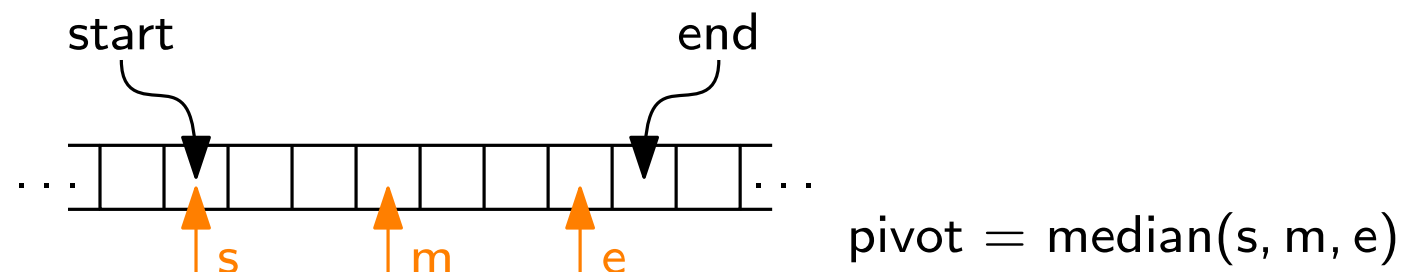
## Common optimizations

- Median-of-3 pivoting

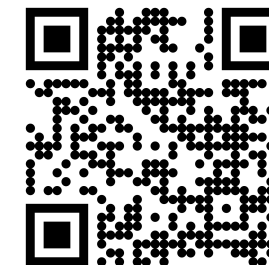
- Adaptive sorting:

If the partition is smaller than  $b$  (for a given parameter  $b$ ) stop the recursion. At the end call insertion sort to finish the mostly sorted array.

$b = 16$



see (and hear) it!



# Advanced Algorithms

## Algorithms in Practice

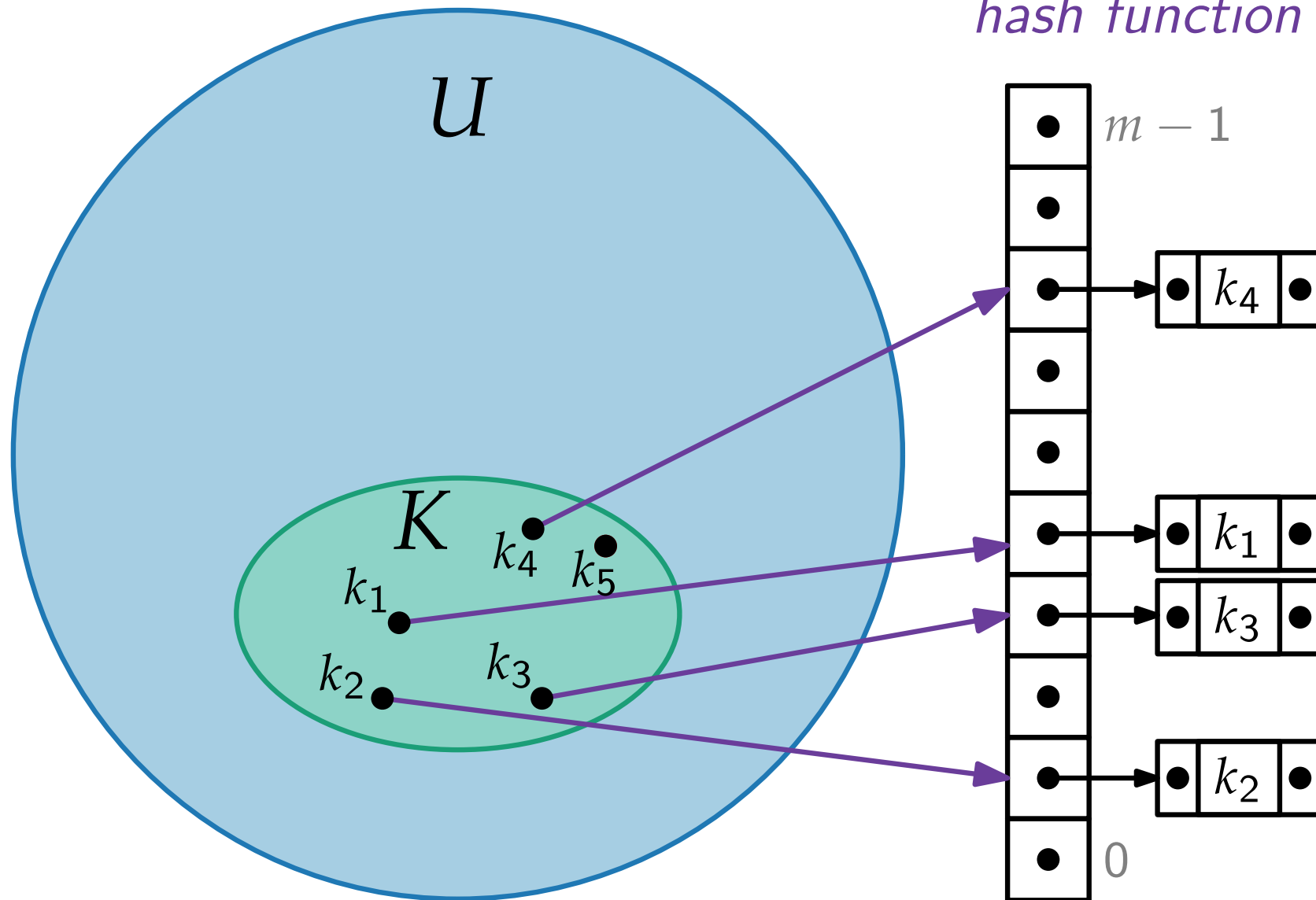
### Hash Maps: A Look Under the Hood

Tim Hegemann · WS23

# Recap: Hash Tables

**Assumption:** large universe  $U$  (that means  $|U| \gg |K|$ )

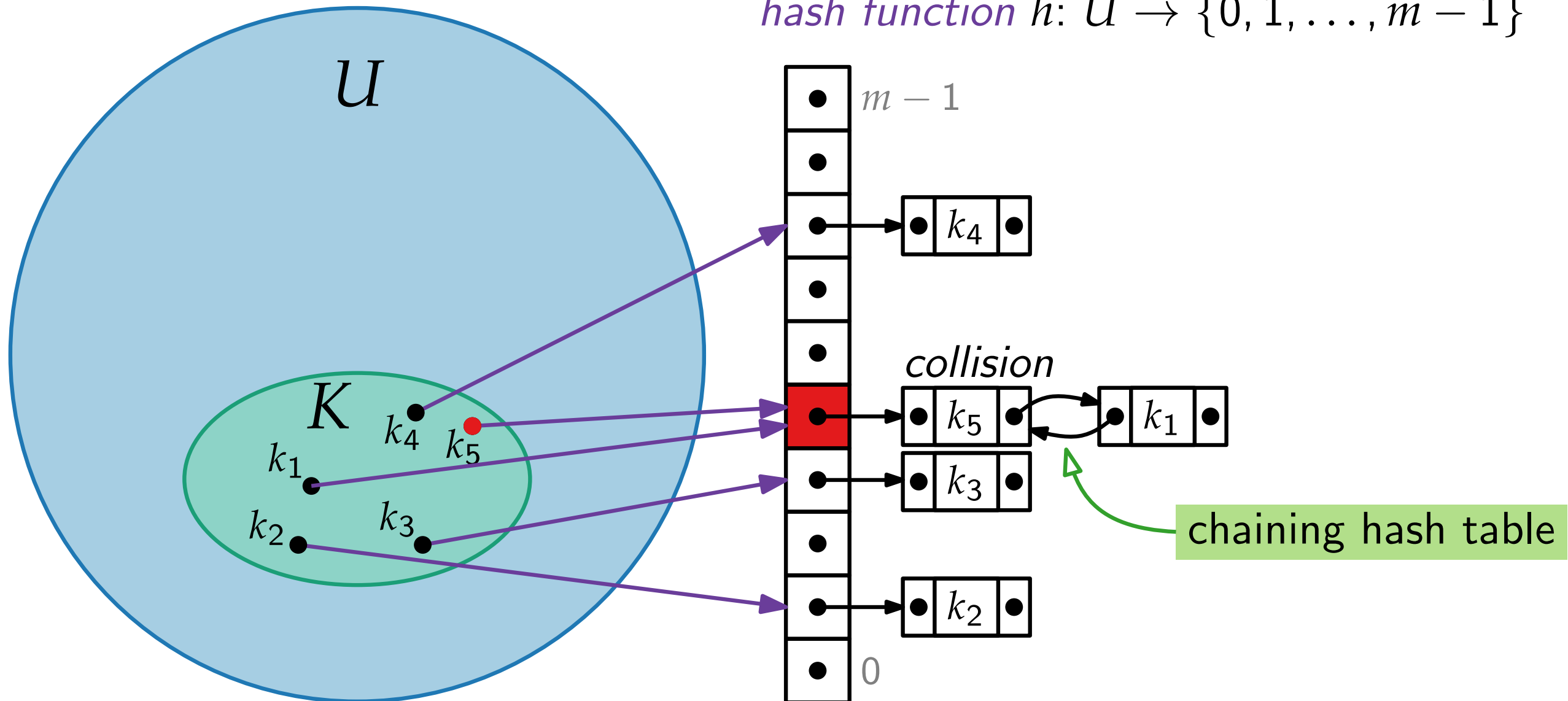
*hash function*  $h: U \rightarrow \{0, 1, \dots, m - 1\}$



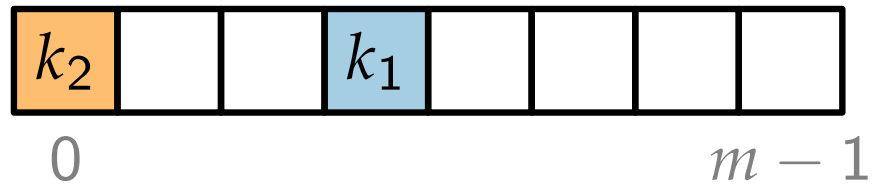
# Recap: Hash Tables

**Assumption:** large universe  $U$  (that means  $|U| \gg |K|$ )

hash function  $h: U \rightarrow \{0, 1, \dots, m - 1\}$



# Probing Hash Table



insert( $k_1$ )

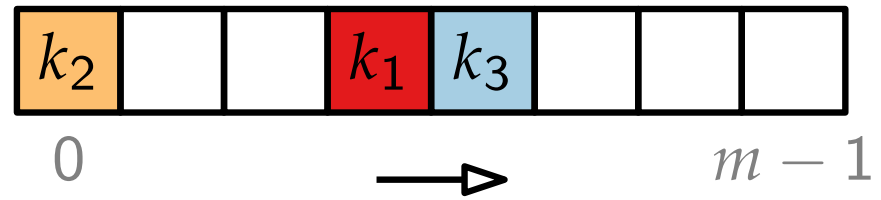
insert( $k_2$ )

$$h(k_1) = 3$$

$$h(k_2) = 0$$



# Probing Hash Table



insert( $k_1$ )

“linear probing”

insert( $k_2$ )

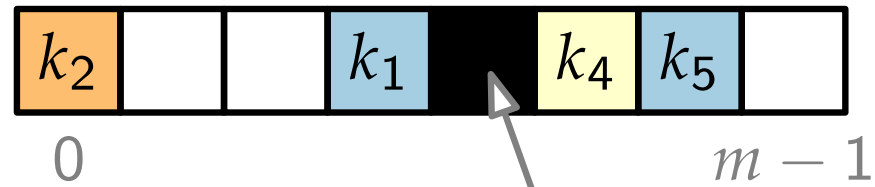
insert( $k_3$ )

$$h(k_1) = 3$$

$$h(k_2) = 0$$

$$h(k_3) = 3$$

# Probing Hash Table



insert( $k_1$ )

insert( $k_2$ )

insert( $k_3$ )

insert( $k_4$ )

insert( $k_5$ )

delete( $k_3$ )



$$h(k_1) = 3$$

$$h(k_2) = 0$$

$$h(k_3) = 3$$

$$h(k_4) = 4$$

$$h(k_5) = 3$$

# Probing Hash Table



insert( $k_1$ )

insert( $k_2$ )

insert( $k_3$ )

insert( $k_4$ )

insert( $k_5$ )

delete( $k_3$ )

including tombstones!

load factor:  $\alpha = \frac{|K|}{m}$

$\alpha > \alpha_{\max}$ ?    Resize & Rebuild!

$h(k_1) = 3$

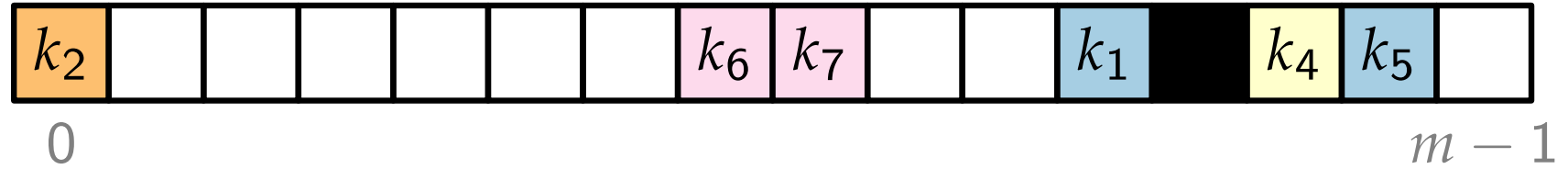
$h(k_2) = 0$

$h(k_3) = 3$

$h(k_4) = 4$

$h(k_5) = 3$

# Swisstable



Ingredients:

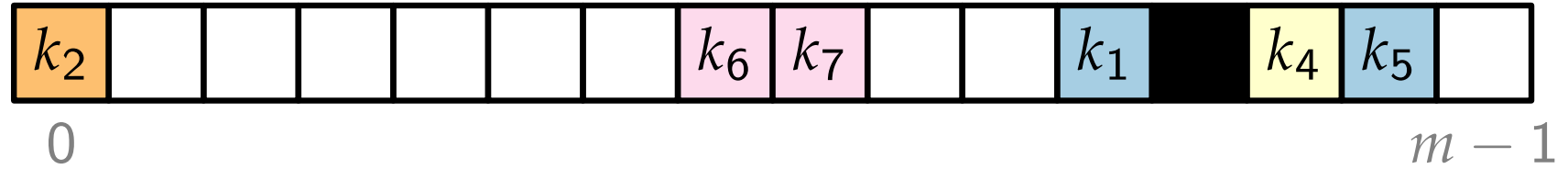
- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

$$h : U \rightarrow \text{u64}$$

$$H_1(k) = h(k) \bmod m$$

$$H_2(k) = h(k) \ggg 57 \quad \text{“first 7 bits”}$$

# Swisstable




Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

$m$  bytes of

$0 \times \text{FF}$  = empty

$0 \times 80$  = deleted 

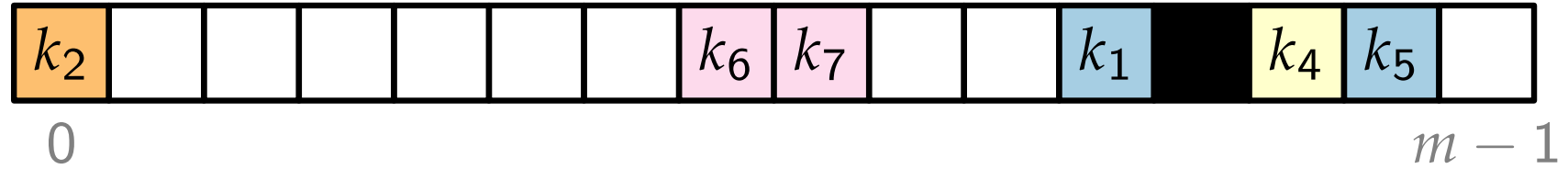
$0 \times 00 \dots 0 \times 7\text{F}$  = occupied

use  $H_2$  for the occupied bytes

+ reprise of the first 16 bytes



# Swisstable



Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

`_mm_cmpeq_epi8`

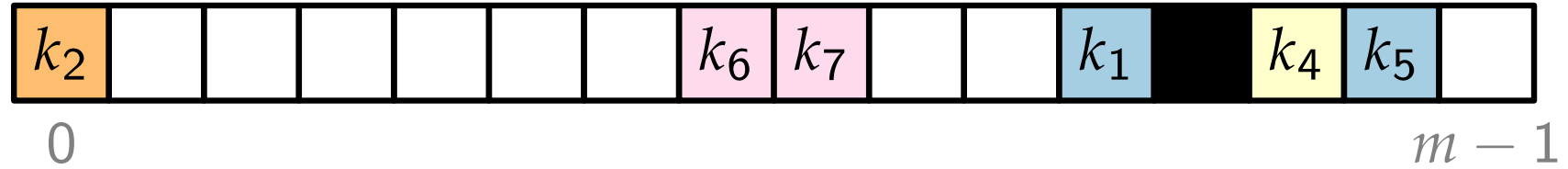
A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1	A1
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

05	7B	A1	82	CF	40	6A	37	42	A1	9B	58	E2	FA	36	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

→

00	00	FF	00	00	00	00	00	00	00	FF	00	00	00	00	00
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

# Swisstable

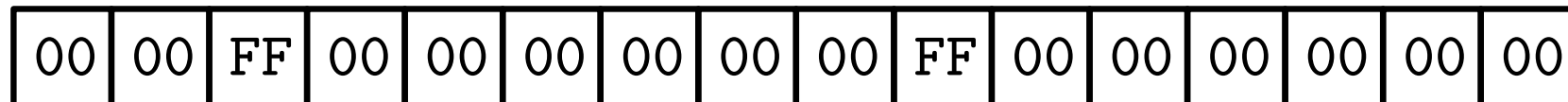


Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

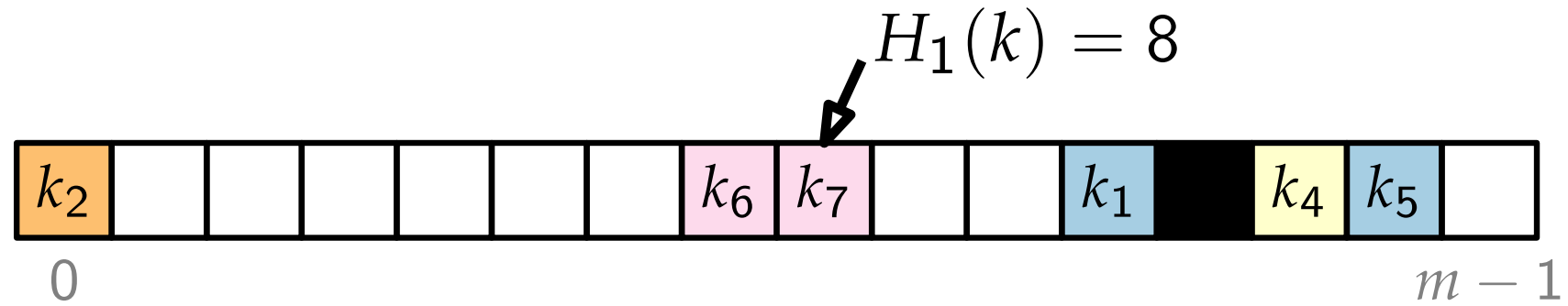
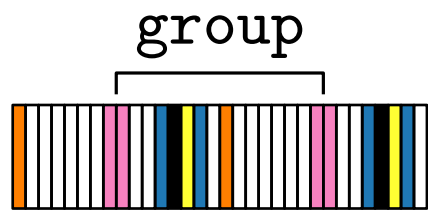
`_mm_movemask_epi8`

$0 \times 2040$





# Swisstable



## Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

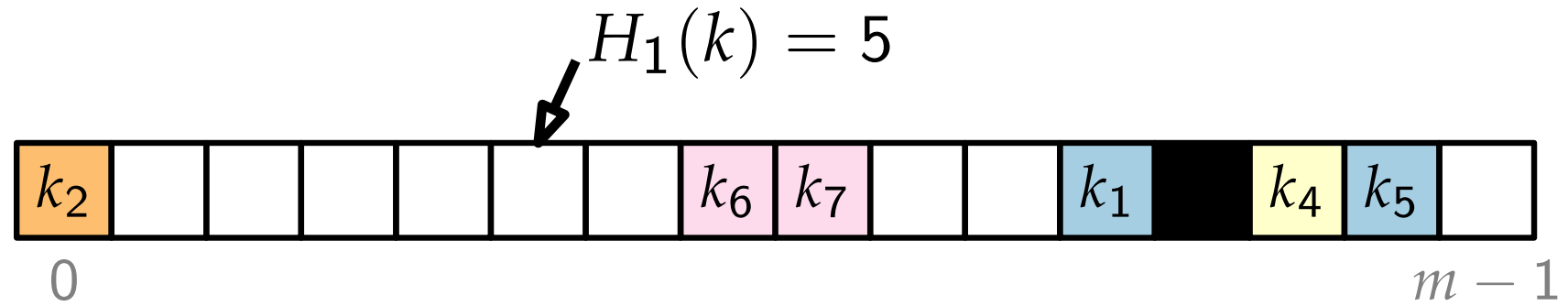
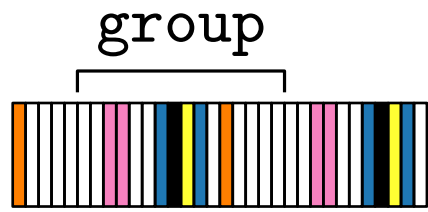
## Operations:

- $\text{find}(k: \text{Key})$
- $\text{insert}(k: \text{Key})$
- $\text{delete}(k: \text{Key})$

## Steps:

1. start at bucket  $H_1(k)$
2. search group for  $H_2(k)$
3. for each match, check keys, return true if found
4. search group for an empty bucket
5. return false if found
6. otherwise, check the next group

# Swisstable



## Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

## Operations:

- $\text{find}(k: \text{Key})$
- $\text{insert}(k: \text{Key})$
- $\text{delete}(k: \text{Key})$

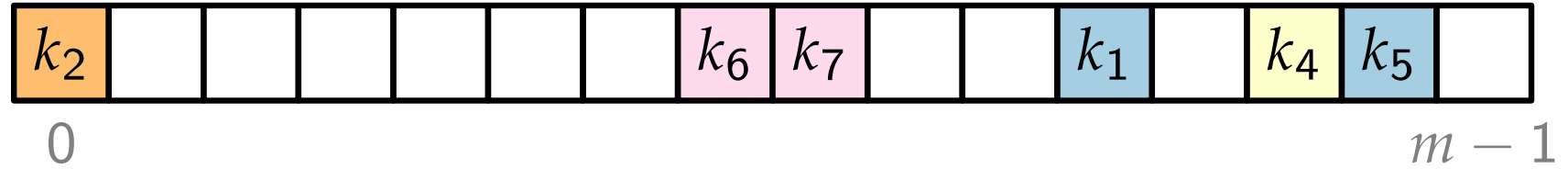
Case 1:  $k$  is in the table

find  $k$ , then replace it

Case 2:  $k$  is not in the table

1. start at bucket  $H_1(k)$
2. search group for empty or deleted
3. if found, insert  $k$
4. otherwise, check the next group

# Swisstable

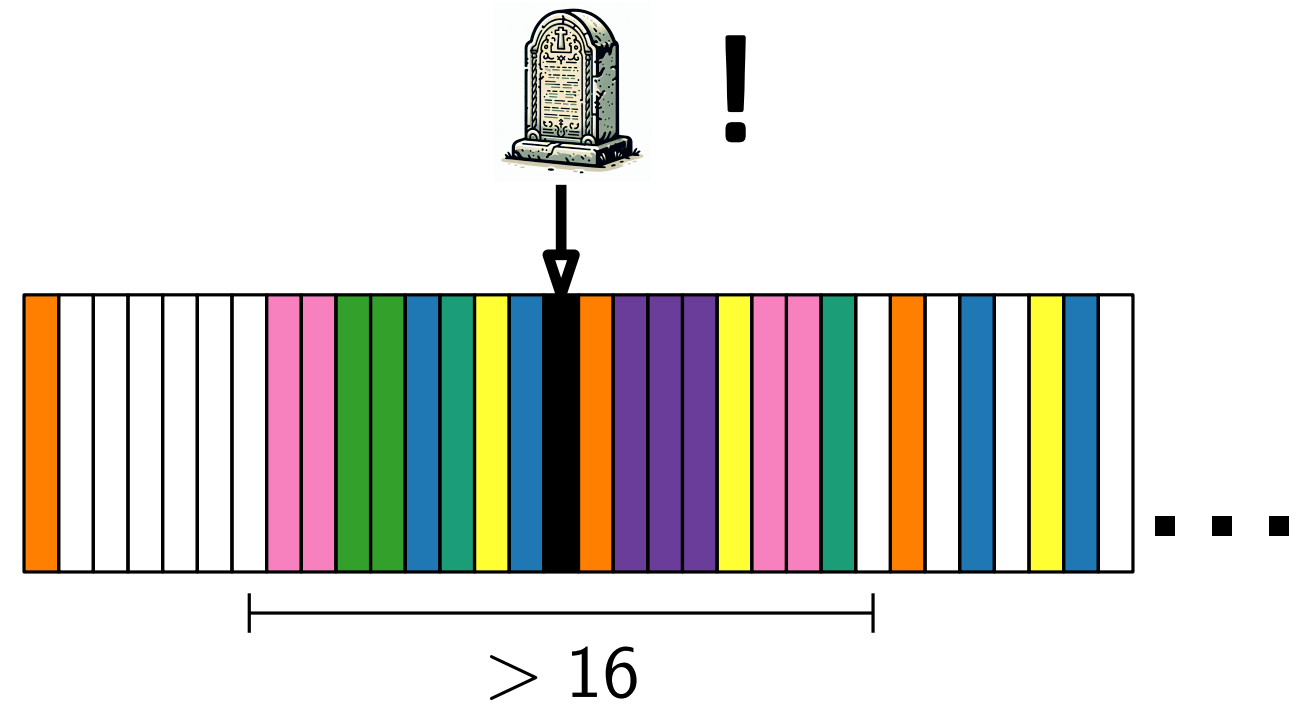


## Ingredients:

- derive two hash functions
- use  $m + 16$  bytes of extra metadata
- SIMD instructions

## Operations:

- $\text{find}(k: \text{Key})$
- $\text{insert}(k: \text{Key})$
- $\text{delete}(k: \text{Key})$



# Further Reading

Slides based on:

- “Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step” by Matt Kulkundis
- “Swisstable, a Quick and Dirty Description” by Aria Beingessner

