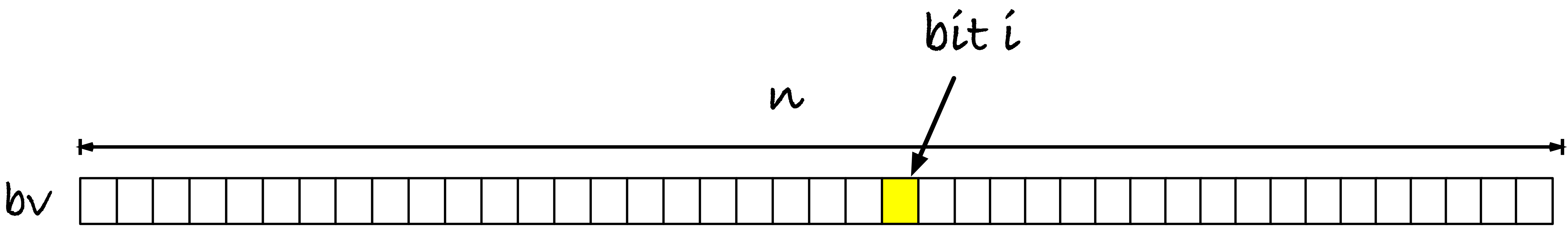
Bit-vectors and rank

Going from O(n log n) bits overhead to ŏ(n) bits

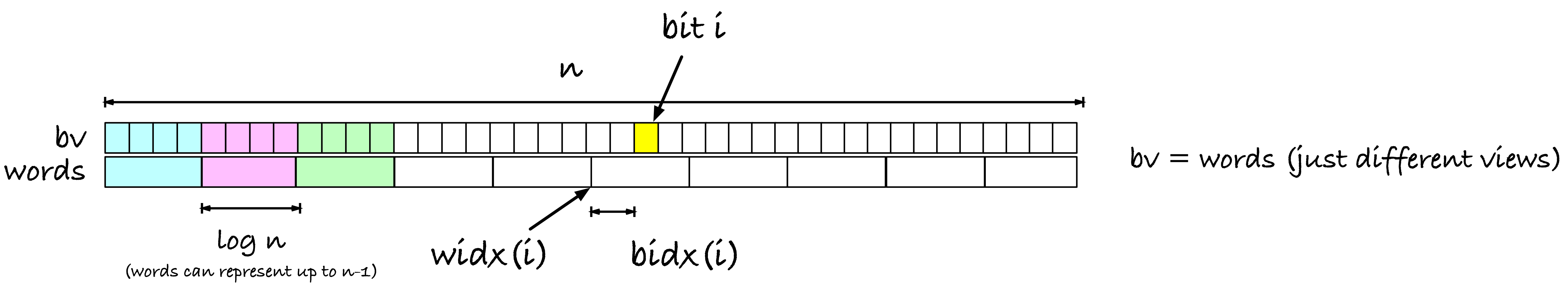
Bit vectors



set bit: bv[i] = b

get bit: bv[i]

Bit vectors




```go
const (
  // 32-bit integers (change as needed)
  wordsize = 32
)

func widx(i uint32) uint32 { return i / wordsize }
func bidx(i uint32) uint32 { return i % wordsize }
```

```go
const (
  wordsize = 32 // 2^5
  divshift = 5
  modmask  = (1 << divshift) - 1
)

// bit versions of i / wordsize and i % wordsize ...
func widx(i uint32) uint32 { return i >> divshift }
func bidx(i uint32) uint32 { return i & modmask }
```

Bit vectors

```
type BitVector struct {
    len   int
    words []uint32
}

func NewBitVector(len int) *BitVector {
   noWords := (len + wordsize - 1) / wordsize
   words := make([]uint32, noWords)
   return &BitVector{len: len, words: words}
}
```

Bit vectors

```python
class BitVector:
    bytes: bytearray
    size: int

    def __init__(self, size: int):
        self.size = size
        self.bytes = bytearray((size+8-1)//8)
```

Integers in Python are not of a fixed size, and it is difficult to work with the underlying computer words… Python isn't the best tool for this job, but we can easily manage bit-vectors as bytearrays.

Bit vectors and bits in words

bv 0 1 0 1

words 1 0 1 0

When we write words, the least-significant bit goes to the right. Move left to go to higher bits, move right to go to lower bits.

When we manipulate words, they are in the opposite order of what we draw for the bit-vector.

Shift left: 1 0 1 1 << 2 = 1 1 0 0

Shift right: 1 0 1 1 >> 2 = 0 0 1 0 (logical: shifts in zeros)

1 0 1 1 >> 2 = 1 1 1 0 (arithmetic: shifts in the sign bit)

AND:
1 0 1 1
1 0 0 1

1 0 0 1

(1 & 1 = 1, zero otherwise)

OR:
1 0 1 1
1 0 0 1

1 0 1 1

(0 | 0 = 0, one otherwise)

XOR:
1 0 1 1
1 0 0 1

0 0 1 0

(1^0 = 0^1 = 1
1^1 = 0^0 = 0)

NEG:
1 0 1 1

0 1 0 0

(flip all the bits)

MINUS:
1 0 1 1

0 1 0 1

(two's complement: -x = ~x+1)

Operators vary from language to language but you typically have them

Masking

Mask bit i:

mask = 0001 << i = 0100

mask & word

0100

0 b_2 0 0

(word & (1 << i)) >> i = 0 0 0 b_2

word = b_3 b_2 b_1 b_0

shift = b_3 b_2 b_1 b_0 >> i = ? ? b_3 b_2

(word >> i) & 1 = ? ? b_3 b_2

0001

= 0 0 0 b_2

Mask k bits:

mask = (0001 << i = 0100) - 1 = 0011

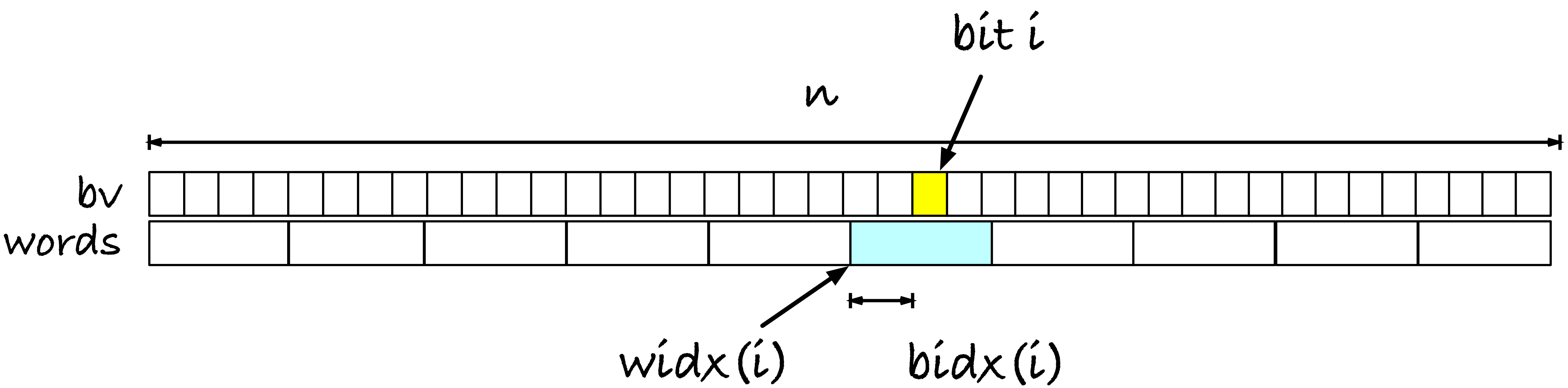
mask & word

b_3 b_2 b_1 b_0

0011

0 0 b_1 b_0

Access



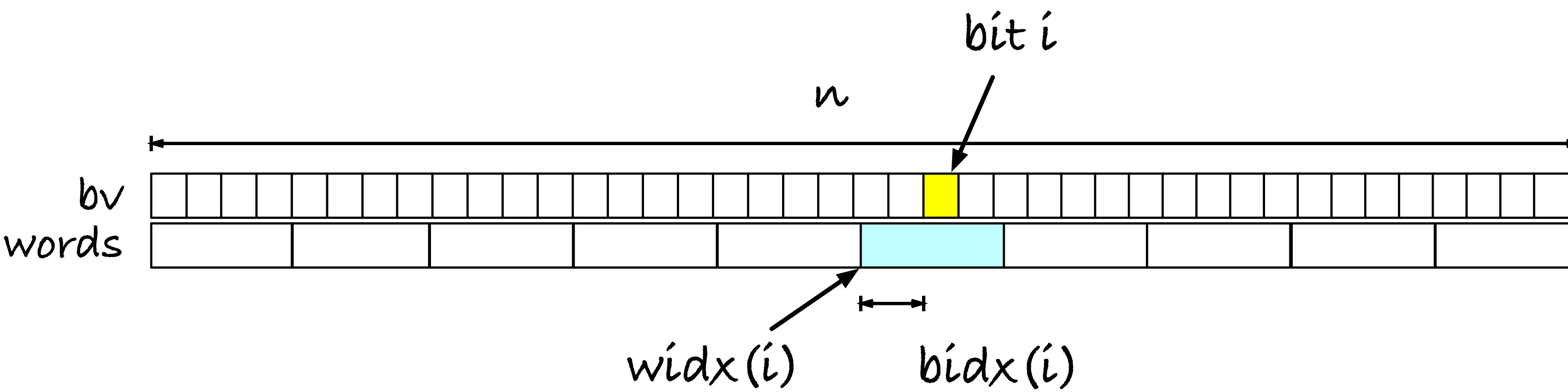
`bv[i]` amounts to:

```
words[widx(i)] =
1 << bidx(i) =
words[widx(i)]&(1<<bidx(i)) =
```

`words[widx(i)] =`	$b_3\ b_2\ b_1\ b_0$
`1 << bidx(i) =`	0 1 0 0
`words[widx(i)]&(1<<bidx(i)) =`	0 b_2 0 0

`(words[widx(i)]&(1<<bidx(i)) ≠ 0) if you want a truth value…`

Setting bits




```
bv[i] = 1: words[widx(i)] |= 1 << bidx(i)

words[widx(i)] =                    b3 b2 b1 b0
1 << bidx(i) =                      0 1 0 0
words[widx(i)] | (1<<bidx(i)) =     b3 1 b1 b0


bv[i] = 0: words[widx(i)] &= ~(1 << bidx(i))

words[widx(i)] =                    b3 b2 b1 b0
1 << bidx(i) =                      0 1 0 0
~(1 << bidx(i)) =                   1 0 1 1
words[widx(i)] & ~(1<<bidx(i)) =    b3 0 b1 b0
```

```python
class BitVector:
    bytes: bytearray
    size: int

    def __init__(self, size: int):
        self.size = size
        self.bytes = bytearray((size+8-1)//8)

    def __getitem__(self, i: int) -> bool:
        return bool(self.bytes[i//8] & (1 << (i % 8)))

    def __setitem__(self, i: int, b: bool) -> None:
        if b:
            self.bytes[i//8] |= (1 << (i % 8))
        else:
            self.bytes[i//8] &= ~(1 << (i % 8))

    def __len__(self) -> int:
        return self.size
```

```go
type BitVector struct {
    len   uint32
    words []uint32
}

func NewBitVector(len uint32) *BitVector {
   noWords := (len + wordsize - 1) / wordsize
   words := make([]uint32, noWords)
   return &BitVector{len: len, words: words}
}

func (bv *BitVector) Len() uint32 { return bv.len }

func (bv *BitVector) Set(i uint32, b bool) {
    if b {
        bv.words[widx(i)] |= 1 << bidx(i)
    } else {
        bv.words[widx(i)] &= ^(1 << bidx(i))
    }
}

func (bv *BitVector) Access(i uint32) bool {
    return bv.words[widx(i)]&(1<<bidx(i)) != 0
}
```

Rank

n

bv:	0	1	1	0	0	0	1	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	1	1	0	0	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0
$rank_0(i)$:	0	1	1	1	2	3	4	4	5	5	5	5	5	6	6	7	…																								
$rank_1(i)$:	0	0	1	2	2	2	2	3	3	4	5	6	7	7	8	8	…																								

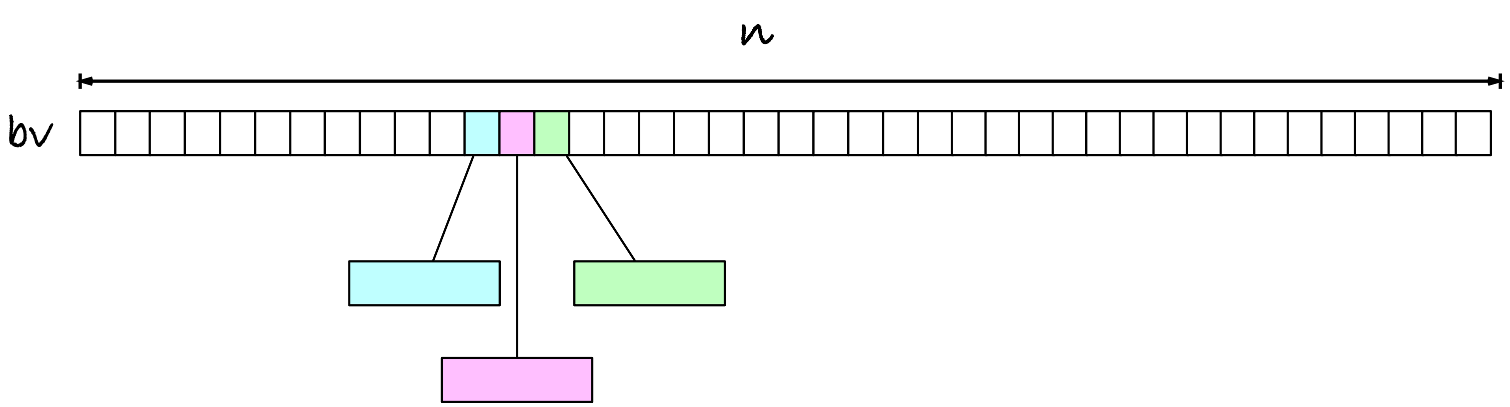
$rank_0(i) = |\{j \mid bv[j] = 0; j < i\}|$

$rank_1(i) = |\{j \mid bv[j] = 1; j < i\}|$

$rank_0(i) = i - rank_1(i)$

Suffices to handle $rank_1(i)$

Naïve solution



$\text{rank}_1(i) = \text{acc}[i] = |\ \{j \mid bv[j] == 1; j < i\}\ |$

acc[0] = 0

acc[i] = acc[i-1] + bv[i-1]

Build: Scan through bv, O(n) time.

Query: lookup acc[i], O(1) time.

Space acc[i] = log n bits

Space acc = n log n bits

Word rank

popcount(w) = number of 1-bits set in a word
(this is usually an instruction on the machine)

w:	1	0	1	1	0	1	0	1

popcount(w) = 5

Rank for the first k bits is the popcount for those k bits. Call it wrank(w,k). How do we compute wrank(w,k)?

mask(k) = (1 << k) - 1
mask(0) = 0b00000001 - 1 = 0b00000000
mask(1) = 0b00000010 - 1 = 0b00000001
mask(2) = 0b00000100 - 1 = 0b00000011
mask(3) = 0b00001000 - 1 = 0b00000111
mask(4) = 0b00010000 - 1 = 0b00001111
...

w:	1	0	1	1	0	1	0	1
mask(0):	0	0	0	0	0	0	0	0
w & mask(0):	0	0	0	0	0	0	0	0

popcount(w & mask(0)) = 0

w:	1	0	1	1	0	1	0	1
mask(2):	0	0	0	0	0	0	1	1
w & mask(0):	0	0	0	0	0	0	0	1

popcount(w & mask(2)) = 1

w:	1	0	1	1	0	1	0	1
mask(2):	0	0	0	0	1	1	1	1
w & mask(0):	0	0	0	0	0	1	0	1

popcount(w & mask(4)) = 2

wrank(w, k) = popcount(w & mask(k))

Word rank

```
; int32_t wrank(int32_t w, int32_t k)
; System V AMD64 ABI so w in edi and k in esi.
; Result is returned in eax.
wrank:
    mov eax, 1          ; get the mask
    mov ecx, esi
    shl eax, cl         ; (1 << k)
    dec eax             ;          - 1

    and eax, edi        ; apply mask
    popcnt eax, eax     ; then popcount
    ret                 ; and we're done
```

Word rank

```c
#include <stdint.h>

uint32_t wrank(uint32_t w, uint32_t k) {
    uint32_t mask = (1 << k) - 1;
    return __builtin_popcount(w & mask);
}
```

__builtin_popcount is a complier extension in gcc and clang.

Word rank

```
def wrank(w : int, k : int) -> int:
    mask = (1 << k) - 1
    return bin(w & mask).count("1")

# Python 3.10+
def wrank(w : int, k : int) -> int:
    mask = (1 << k) - 1
    return (w & mask).bit_count()
```

Word rank

```
func wrank(w uint32, k uint32) uint32 {
    var mask uint32 = (1 << k) - 1
    return uint32(bits.OnesCount32(w & mask))
}
```

Popcount

```
cdef inline int popcount(uint32_t w):
    # Wegner/Kernigan method
    cdef int count = 0
    while w:
        count += 1
        w &= w - 1 # masking out the rightmost bit
    return count
```

```
          w = 0b0010110100010000
      w - 1 = 0b0010110100001111
  w & (w-1) = 0b0010110100000000
```

w &= w - 1 removes the right-most bit in w.

Popcount

Counts for all possible bytes.
We can compute these ahead of time and compile the table into our program.

```
cdef unsigned char pcnt[256]
cdef popcount1(uint32_t w):
    return pcnt[w&0xff] + pcnt[(w >> 8) & 0xff] + \
           pcnt[(w >> 16) & 0xff] + pcnt[w >> 24]
```

Mask and shift to get the four bytes in a 32 bit word, and add the tabulated values.

Popcount

```
DEF MASK_10_11   = 0x55555555 # 0b0101010101010101…
DEF MASK_20_21   = 0x33333333 # 0b0011001100110011…
DEF MASK_40_41   = 0x0f0f0f0f # 0b0000111100001111000…
DEF MASK_80_81   = 0x00ff00ff # 0b000000001111111110000…
DEF MASK_160_161 = 0x0000ffff # 0b00000000000000001111…

cdef popcount(uint32_t w):
    w = (w & MASK_10_11) + ((w >> 1) & MASK_10_11)
    w = (w & MASK_20_21) + ((w >> 2) & MASK_20_21)
    w = (w & MASK_40_41) + ((w >> 4) & MASK_40_41)
    w = (w & MASK_80_81) + ((w >> 8) & MASK_80_81)
    return (w & MASK_160_161) + ((w >> 16) & MASK_160_161)
```

Example with 8 bit words!

```
Think of w as a vector of 8 1-bit words:
            w = [0][0][1][0][1][1][0][1]

We are going to add the bits pairwise…

w & m1 picks the numbers at even offsets.
(w >> 1) & m1 picks the numbers at odd offsets.

      w =  00  10  11  01        w >> 1 =   0  01  01  10
     m1 =  01  01  01  01            m1 =  01  01  01  01
      & = [00][00][01][01]            & = [00][01][01][00]

Add them, effectively adding them as 2-bit numbers, and
you get the popcount in slices of length two.

 w = [00][10][11][01]                w & m1 = [00][00][01][01]
                              (w >> 1) & m1 = [00][01][01][00]
                                            = [00][01][10][01]
                                            =   0   1   2   1
```

Example with 8 bit words!

```
Original w: [00][10][11][01]
 Current w: [00][01][10][01]
          = [ 0][ 1][ 2][ 1]
```

```
We are going to add 2-bit slices pairwise…

w & m2 picks the 2-bit slices at even offsets.
(w >> 2) & m2 picks the slices at odd offsets.
```

```
   w =  0001  1001             w >> 2 =  0000  0110
  m2 =  0011  0011                 m2 =  0011  0011
   & = [0001][0001]                 & = [0000][0010]
```

```
Add them, effectively adding them as 4-bit numbers, and
you get the popcount in slices of length four.
```

```
                                     w & m2 = [0001][0001]
                              (w >> 2) & m2 = [0000][0010]
  Original w: [0010][1101]                  = [0001][0011]
                                            =      1     3
```

Example with 8 bit words!

```
Original w: [0010][1101]
 Current w: [0001][0011]
          = [   1][   3]
```

We are going to add the 2 four-bit slices…

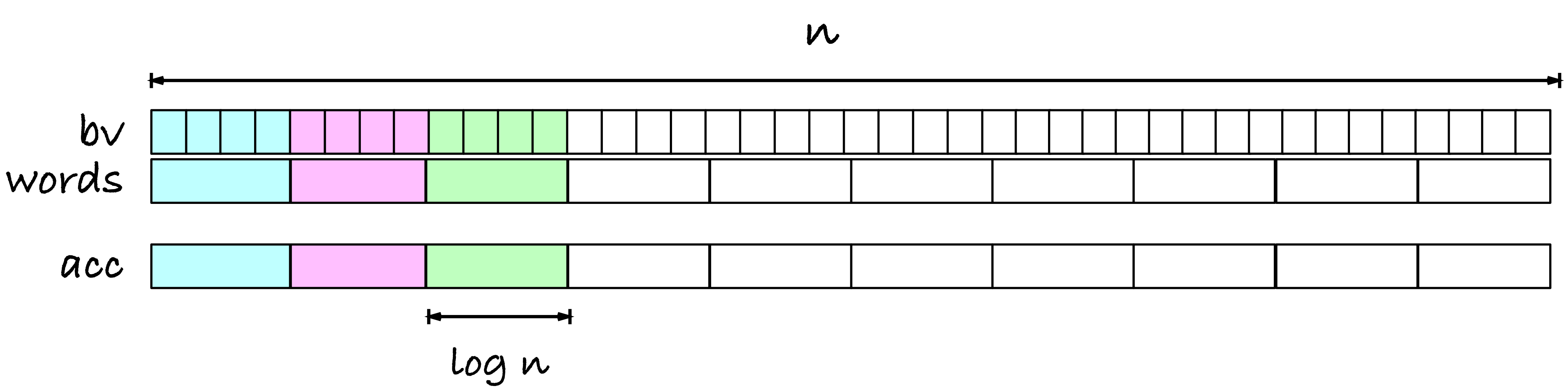
w & m4 picks the 4-bit slices at even offsets.
(w >> 4) & m4 picks the slices at odd offsets.

```
  w =  00010011          w >> 4 =  00000001
 m4 =  00001111              m4 =  00001111
  & = [00000011]              & = [00000001]
```

Add them, effectively adding them as 4-bit numbers, and you get the popcount in slices of length eight.

```
                                  w & m4 = [00000011]
                           (w >> 4) & m4 = [00000001]
Original w: [00101101]                   = [00000100]
                                         =          4
```

Rank with words…



$acc[i]$ = number of 1s in blocks 0, …, i-1

Space acc = $\lceil n / (\log n) \rceil \log n$ = n bits

$rank_1(i) = acc[widx(i)] + wrank_1(words[widx(i)], bidx(i))$

```go
type BitVector struct {
    len         uint32
    words       []uint32
    wordAccRank []uint32
}

func NewBitVector(len uint32) *BitVector {
    noWords := (len + wordsize - 1) / wordsize
    words := make([]uint32, noWords+1) // + 1 to index past end
    acc := make([]uint32, noWords+1)   // + 1 to index past end
    return &BitVector{len: len, words: words, wordAccRank: acc}
}

func (bv *BitVector) Len() uint32           { … }
func (bv *BitVector) Set(i uint32, b bool) { … }
func (bv *BitVector) Access(i uint32) bool { … }
```

```go
func (bv *BitVector) PreprocessRank() {
    var acc uint32 = 0
    for i := range bv.words {
        bv.wordAccRank[i] = acc
        acc += uint32(bits.OnesCount32(bv.words[i]))
    }
}

func wrank(w uint32, k uint32) uint32 {
    var mask uint32 = (1 << k) - 1
    return uint32(bits.OnesCount32(w & mask))
}

func (bv *BitVector) Rank0(i uint32) uint32 {
    return i - bv.Rank1(i)
}
func (bv *BitVector) Rank1(i uint32) uint32 {
    return bv.wordAccRank[widx(i)] +
              wrank(bv.words[widx(i)], bidx(i))
}
```

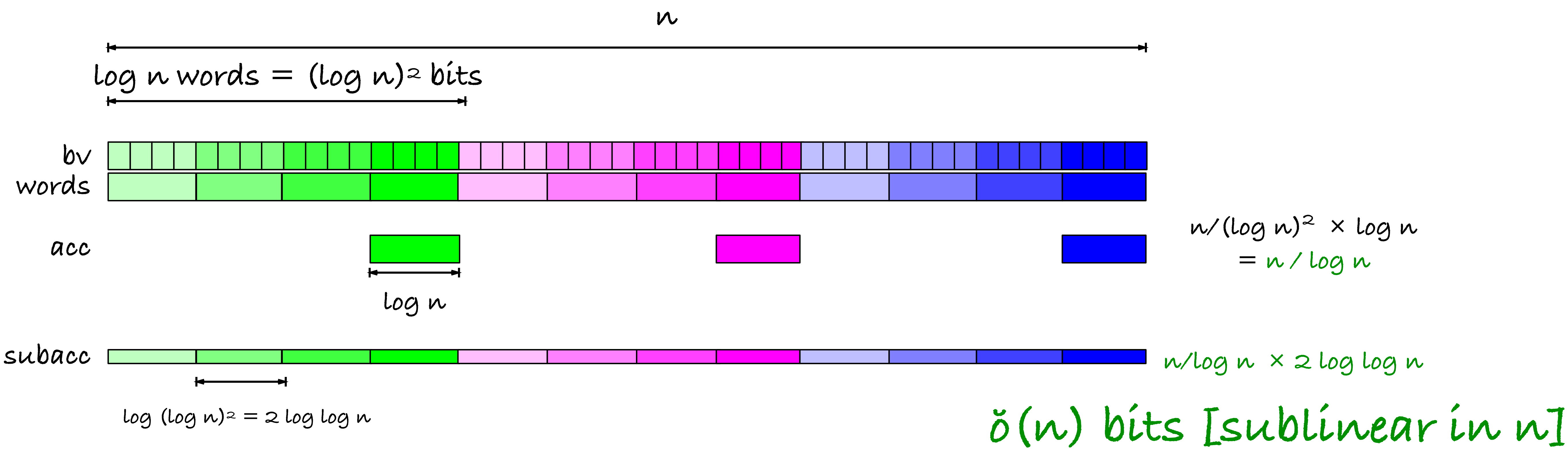
```
from libc.stdint cimport uint32_t
from cpython.mem cimport PyMem_Malloc, PyMem_Free

# The generated C code is more complex than /WORDSIZE and %WORDSIZE
# so I don't trust the compiler to make them into bit operations...
DEF WORDSIZE   = 32
DEF WORD_SHIFT = 5
DEF WORD_MASK  = ((1 << WORD_SHIFT) - 1)

cdef inline uint32_t widx(uint32_t i): return i >> WORD_SHIFT
cdef inline uint32_t bidx(uint32_t i): return i & WORD_MASK
cdef popcount(uint32_t w): … # one of those from earlier

cdef inline int wrank(uint32_t w, uint32_t i):
    return popcount(w & ((1 << i) - 1))
```

```
cdef class BitVector:
    cdef:
        uint32_t size, no_words
        uint32_t *words
        uint32_t *acc

    def __cinit__(self, uint32_t size):
        self.size = size
        # The +1 to get indexing past the last bit
        self.no_words = (size + WORDSIZE - 1) / size + 1

        self.words = \
           <uint32_t *> PyMem_Malloc(self.no_words * sizeof(self.words[0]))
        if not self.words: raise MemoryError()
        self.acc = \
           <uint32_t *> PyMem_Malloc(self.no_words * sizeof(self.acc[0]))
        if not self.acc: raise MemoryError()

        # clear the words. Worry about acc when we preprocess.
        cdef uint32_t i
        for i in range(self.no_words):
            self.words[i] = 0

    def __dealloc__(self):
        PyMem_Free(self.words)
        PyMem_Free(self.acc)
```

```
def __len__(self):
    return self.size

def __getitem__(self, uint32_t i):
    # return as an integer, zero or one, by shifting back again
    return int((self.words[widx(i)] & (1 << bidx(i))) >> bidx(i))

def __setitem__(self, uint32_t i, bint b):
    if b:
        self.words[widx(i)] |= (1 << bidx(i))
    else:
        self.words[widx(i)] &= ~(1 << bidx(i))
```

```
def preprocess_rank(self):
    cdef uint32_t acc = 0
    cdef uint32_t i
    for i in range(self.no_words):
        self.acc[i] = acc
        acc += popcount(self.words[i])

def rank(self, bint b, uint32_t i):
    if b == 0:
        return i - self.rank(1, i)
    else:
        return self.acc[widx(i)] + wrank(self.words[widx(i)], bidx(i))
```

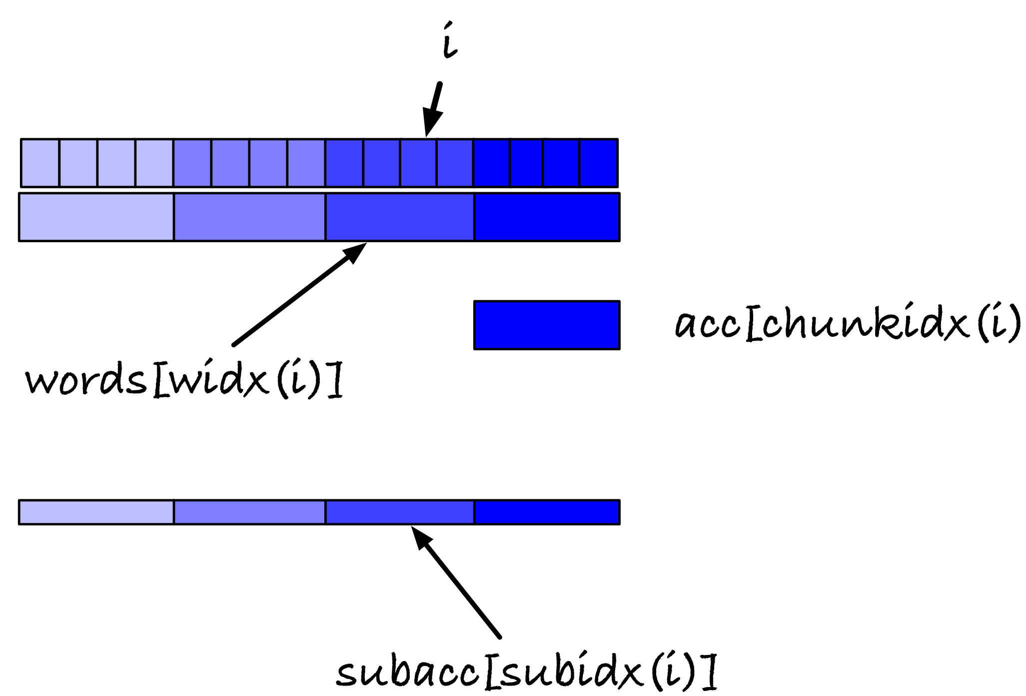
Smaller still...



$$\mathrm{rank}_1(i) = \mathrm{acc}[\mathrm{chunkidx}(i)] + \mathrm{subacc}[\mathrm{subidx}(i)] + \mathrm{wrank}(\mathrm{words}[\mathrm{widx}(i)], \mathrm{bidx}(i))$$

subidx(i) = widx(i) [just smaller type]

$$\mathrm{chunkidx}(i) = i \,/\, (\log n)^2$$



Smaller still…

- word size 32 bits (log n = 32)
  - $(\log n)^2 = 1024$
- $32 = 2^5$ so log log n = 5 and 2 log log n = 10
  - this is not a word size; have to use 16 bit words here
  - that would also work for 64 bit words (2 log 64 = 12)
  - or 128 bit words (2 log 128 = 14)

Smaller still…

- Human chromosome 1: $n = 248{,}956{,}422$, call it ~256×10^6 bits or 32 megabytes.
  - When you see log n, think 32 for 32-bit integers. Then 2 log log n requires 16 bit words
- First solution:
  - Overhead for acc: same as bits, so 32Mb.
- Second solution:
  - Overhead for acc:
    - n/log n = 32 Mb divided by 32, so 1 Mb
  - Overhead for sub-acc: n/log n × 2 log log n = 1Mb × 16 = 16 Mb
  - Combined: 17 Mb, so a bit more than half the space usage…

```go
const (
    wordSize      = 32                    // ≈ log n
    smallWordSize = 16                    // 2 log log n (rounded up)
    chunkSize     = wordSize * wordSize // (log n)² = 1,024 bits (32 words)
    wordsPerChunk = wordSize              // log n words per chunk
)

// I am relying on Go converting division to shift and remainder
// to masks. The assembly generated bears this out...
// I will not need to access bits more than 32 positions into a word
// so the bidx just gives me a byte.
func widx(i uint32) uint32     { return i / wordSize }
func bidx(i uint32) uint8      { return uint8(i % wordSize) }
func chunkIdx(i uint32) uint32 { return i / chunkSize }

func wrank(w uint32, k uint8) uint32 {
    var mask uint32 = (1 << k) - 1
    return uint32(bits.OnesCount32(w & mask))
}
```

```go
type BitVector struct {
    len      uint32
    words    []uint32 // words we store our bits in
    smallacc []uint16 // smaller words for chunk-counting
    largeacc []uint32 // accumulation from chunk to chunk
}

func NewBitVector(len uint32) *BitVector {
    noWords := (len + wordSize - 1) / wordSize
    noChunks := (len + chunkSize - 1) / chunkSize

    words := make([]uint32, noWords+1)      // + 1 so we can index one past
    smallacc := make([]uint16, noWords+1)  // half as many bits as words
    largeacc := make([]uint32, noChunks+1) // 1/log n as many bits as words

    return &BitVector{
        len:      len,
        words:    words,
        smallacc: smallacc,
        largeacc: largeacc}
}
```

```go
func (bv *BitVector) PreprocessRank() {
    var (
        largeacc uint32 = 0
        acc      uint16 = 0
    )

    for i := range bv.words {
        if i%wordsPerChunk == 0 {
            largeacc += uint32(acc)
            bv.largeacc[i/wordsPerChunk] = largeacc
            acc = 0 // reset sub-chunk counter...
        }
        bv.smallacc[i] = acc
        acc += uint16(bits.OnesCount32(bv.words[i]))
    }
}

func (bv *BitVector) Rank0(i uint32) uint32 { return i - bv.Rank1(i) }
func (bv *BitVector) Rank1(i uint32) uint32 {
    return bv.largeacc[chunkIdx(i)] +
        uint32(bv.smallacc[widx(i)]) +
        uint32(wrank(bv.words[widx(i)], bidx(i)))
}
```

Smaller still…

- With 32-bit words, 2 log log n = 10, 16-bit words, the overhead is: n × (1/32 (for acc) + 1/2 (for small acc))
- With 64-bit words, 2 log log n = 12, 16-bit words, the overhead is: n × (1/64 + 1/4)
- With 128-bit words, 2 log log n = 14, 16-bit words, the overhead is: n × (1/128 + 1/8)
- With 256-bit words, 2 log log n = 16, 16-bit words, the overhead is: n × (1/256 + 1/16)
- With 512-bit words, 2 log log n = 18, 32-bit words, the overhead is: n × (1/256 + 1/16)
- With 1,024-bit words, 2 log log n = 20, 32-bit words, the overhead is: n × (1/1,024 + 1/32)

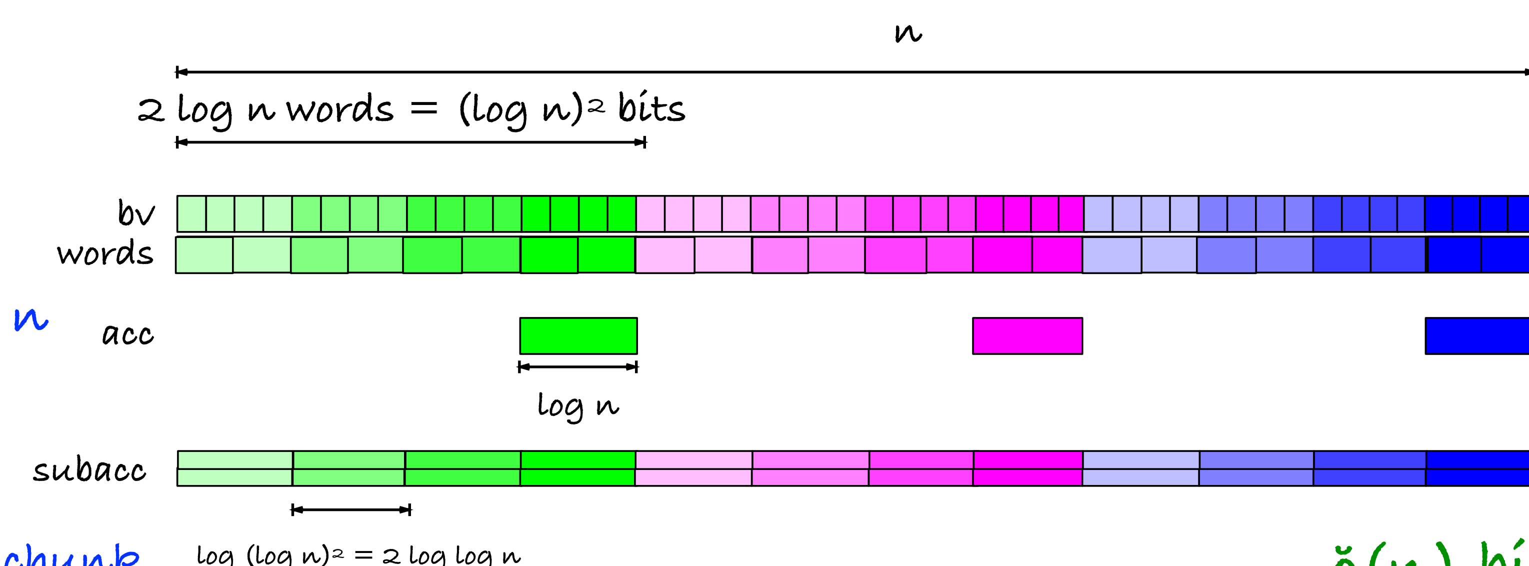
Is there a limit to this?

- This approach works to **obscenely** large n.
- 64-bit words (2 log log n) have log n = $2^{32} \approx 4$ *billion bit* words
  - Those are pretty big words, (more than 500Mb for one word) and we need them for the result so we cannot get rid of that!
  - That is a problem before the method fails!
- $2^{2^{32}} \approx 3 \times 10^{1292913986}$
  - Only 10^{97} elementary particles in the observable universe

Are we done?

- For all practical applications, we are done.
- Theoretically we are not, because
  - Assuming 2log log n is a fixed word size [O(1)] means *everything* is O(1)
- Can we also handle a theoretical (crazy big) n?

Jacobsen's rank



word size ½ log n

2 log n words per chunk

$n/(\log n)^2 \times \log n$
$= n / \log n$

$n/\log n \times 2 \times 2 \log \log n$

$\breve{o}(n)$ bits [sublinear in n]

$\mathrm{rank}_1(i) = \mathrm{acc}[\mathrm{chunkidx}(i)]$
$+ \mathrm{subacc}[\mathrm{subidx}(i)]$
$+ \mathrm{TABLE}[\mathrm{words}[\mathrm{widx}(i)], \mathrm{bidx}(i)]$

Table size for all words of length w: $2^w\, w \log w$

$2^{\frac{1}{2}\log n}\, \frac{1}{2}\log n \log \frac{1}{2}\log n = O(\sqrt{n}\log n \log\log n)$ in $\breve{o}(n)$


That's all Folks!