Improved Address-Calculation Coding of Integer Arrays

Jyrki Katajainen[1,2]

Amr Elmasry[3], **Jukka Teuhola**[4]

[1] University of Copenhagen
[2] Jyrki Katajainen and Company
[3] Alexandria University
[4] University of Turku

Problem formulation

Given: An array of integers $\{x_i \mid i \in \{1, 2, \ldots, n\}\}$

Wanted: Compressed representation, fast random access

Operations:

$access(i)$: retrieve x_i

$insert(i, v)$: insert v before x_i

$delete(i)$: remove x_i

Other: omitted in this talk

$sum(j)$: retrieve $\sum_{i=1}^{j} x_i$

$search(p)$: find the rank of the given prefix sum p

$modify(i, v)$: change x_i to v

Many solutions known, see the list of references in the paper

Theoretical approaches

- $O(1)$ worst-case-time $access$
- overhead of $o(n)$ bits with respect to some measure of compactness
- complicated

Practical approaches

- slower $access$
- $O(n)$ bits of overhead
- implementable
- fast in practice

Measures of compactness

What is optimal?

n: # integers

$\widehat{x} = \max_{i=1}^{n} x_i$

$s = \sum_{i=1}^{n} x_i$

Data-aware measure

Raw representation:

$\sum_{i=1}^{n} \lceil \lg(1 + x_i) \rceil$ bits

Overhead: In order to support random access we expect to need some more bits

Data-independent measures

Compact representation:

$n \lg(1 + s/n) + O(n)$ bits

Apply Jensen's inequality to the raw representation and accept a linear overhead

Lower bound$_1$: $\lceil \lg \widehat{x}^n \rceil$

$\widehat{x}^n$: The number of sequences of n positive integers whose value is at most $\widehat{x}$

Lower bound$_2$: $\left\lceil \lg \binom{s-1}{n-1} \right\rceil$

$\binom{s-1}{n-1}$: The number of sequences of n positive integers that add up to s

Two trivial “solutions”

Uncompressed array

w: size of a machine word

Space: $w \cdot n + O(w)$ bits

access(i): $a[i]$

Access times on my computer:

n	sequential	random
2^{10}	0.89	1.1
2^{15}	0.74	1.4
2^{20}	0.89	7.1
2^{25}	0.74	10.9

↖ ns per operation

– no compression

+ fast

Fixed-length coding

$\hat{x} = \max_{i=1}^{n} x_i$

$\beta = \lceil \lg(1 + \hat{x}) \rceil$

Space: $\beta \cdot n + O(w)$ bits

access(i):

- compute the word address
- read one or two words
- mask the bits needed

– one outlier ruins the compactness

+ relatively fast

Q: How would you support *insert* and *delete* for these structures?

Two examples

$x_1 = n$, $x_i = 1$ for $i \in \{2, \ldots, n\}$

Raw representation:

$n + O(\lg n)$ bits

Fixed-length coding:

$n \lceil \lg(1 + n) \rceil$ bits

Lower bound$_1$:

$\lceil n \lg n \rceil$ bits

$x_1 = n^2$, $x_i = 1$ for $i \in \{2, \ldots, n\}$

Raw representation:

$n + O(\lg n)$ bits

Compact representation:

$n \lg n + \Theta(n)$ bits

Lower bound$_1$:

$\lceil 2n \lg n \rceil$ bits

Lower bound$_2$:

$n \lg n + \Theta(n)$ bits

N.B. All our representations are compact,
but we do not claim them to be optimal

Our contribution

Teuhola 2011

Interpolative coding of integer sequences supporting log-time random access, *Inform. Process. Manag.* **47**,5, 742–761

Space: $n \lg(1 + s/n) + O(n)$ bits, i.e. compact

access: $O(\lg(n+s))$ worst-case time

insert, *delete*: not supported

This paper

Space: $n \lg(1 + s/n) + O(n)$ bits, i.e. compact

access: $O(\lg\lg(n+s))$ worst-case time in the static case and $O(\lg n)$ worst-case time in the dynamic case

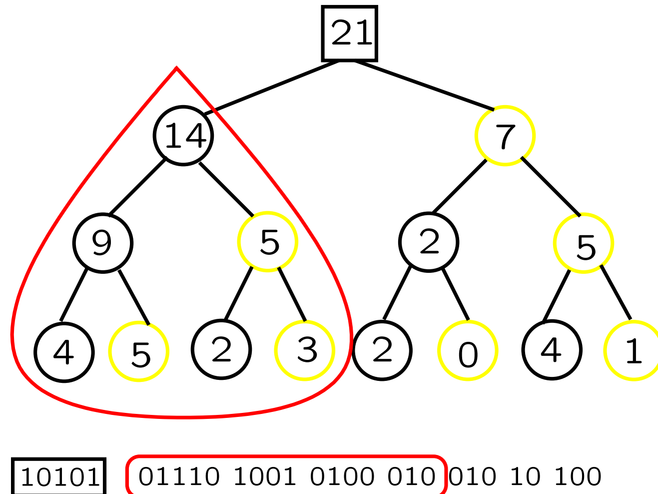
insert, *delete*: $O(\lg n + w^2)$ worst-case time

n: # integers (assume $n \geq w$)

s: sum of the integers

w: size of a machine word

Address-calculation coding



- encoding in depth-first order
- yellow nodes not stored
- skip subtrees using the formula

Space: Compact by the magical formula

access: $O(\lg n)$ worst-case time (assuming that the position of the most significant one bit in a word can be determined in $O(1)$ time)

insert, *delete*: not supported

$$t = \lceil \lg(1 + s) \rceil$$

Magical formula

$$B(n, s) = \begin{cases} n(t - \lg n + 1) + \lfloor \frac{s(n-1)}{2^{t-1}} \rfloor - t - 1 & \text{, if } s \geq n/2 \\ 2^t + \lfloor s(2 - \frac{1}{2^{t-1}}) \rfloor - t - 1 + s(\lg n - t) & \text{, otherwise} \end{cases}$$

Indexed address-calculation coding

c: a tuning parameter, $c \geq 1$
s_i: sum of the numbers in the ith chunk

index; fixed-length coding

chunk size: $k = \lfloor c \cdot \lg(n+s) \rfloor$
chunks: $t = \lceil n/k \rceil$
root: $\lceil \lg(1+s) \rceil$ bits
pointer: $\lg n + \lg\lg(1+s/n) + O(1)$ bits

chunks; address-calculation coding

Analysis

roots:

$\lceil n/k \rceil \cdot \lceil \lg(1+s) \rceil \leq n/c + O(w)$

pointers:

$\lceil n/k \rceil \cdot (\lg n + \lg\lg(1+s/n) + O(1)) \leq n/c + O(w)$

chunks:

$\sum_{i=1}^{t} [k \cdot \lg(1+s_i/k) + O(k)] \leq n \lg(1+s/n) + O(n)$

© Performance Engineering Laboratory

Other applications of indexing

Indexed Elias delta coding

c: a tuning parameter, $c \geq 1$

index; fixed-length coding

chunk size: $k = \lfloor c \cdot (\lg n + \lg\lg s) \rfloor$
chunks: $t = \lceil n/k \rceil$
pointer: $\lg n + \lg\lg(1 + s/n) + O(1)$ bits

chunks; Elias delta coding

Space: raw + $O(\sum_{i=1}^{n} \lg\lg x_i)$

access: $O(\lg n + \lg\lg s)$ worst-case time

Indexed fixed-length coding

c: a tuning parameter, $c \geq 1$

$\hat{x} = \max_{i=1}^{n} x_i$

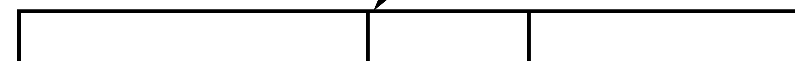
index; fixed-length coding

chunk size: $k = \lfloor c \cdot (\lg n + \lg\lg \hat{x}) \rfloor$
chunks: $t = \lceil n/k \rceil$
pointer: $\lg n + \lg\lg(1 + \hat{x}) + O(1)$ bits

offsets; fixed-length coding

data; raw coding

landmark + offset



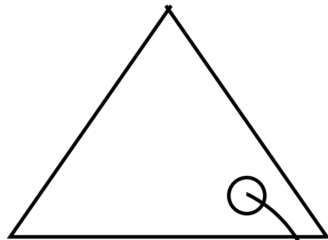
Space: raw + $O(n \lg\lg(n + \hat{x}))$

access: $O(1)$ worst-case time

Dynamization

c: a tuning parameter, $c \geq 1$
w: size of a machine word

index; balanced search tree



chunk size: $k = cw/2..2cw$
chunks: $t = \lceil n/(2cw) \rceil .. \lceil 2n/(cw) \rceil$
root: w bits
pointer: w bits

chunks; address-calculation coding

Use the zone technique:

- align chunks to word boundaries
- keep chunks of the same size in separate zones
- only w zones
- maintain zones as rotated arrays (one chunk may be split)

Space: Still compact

access: $O(\lg n)$ worst-case time ($n \geq w$))

insert, *delete*: $O(\lg n + w^2)$ worst-case time

Experimental setup

Benchmark data:

n integers

– uniformly distributed

– exponentially distributed

Repetitions:

Each experiment repeated r times for sufficiently large r

Reported value:

Measurement result divided by $r \times n$

Processor:

Intel® Xeon® CPU 1.8 GHz $\times$ 2

Programming language:

C

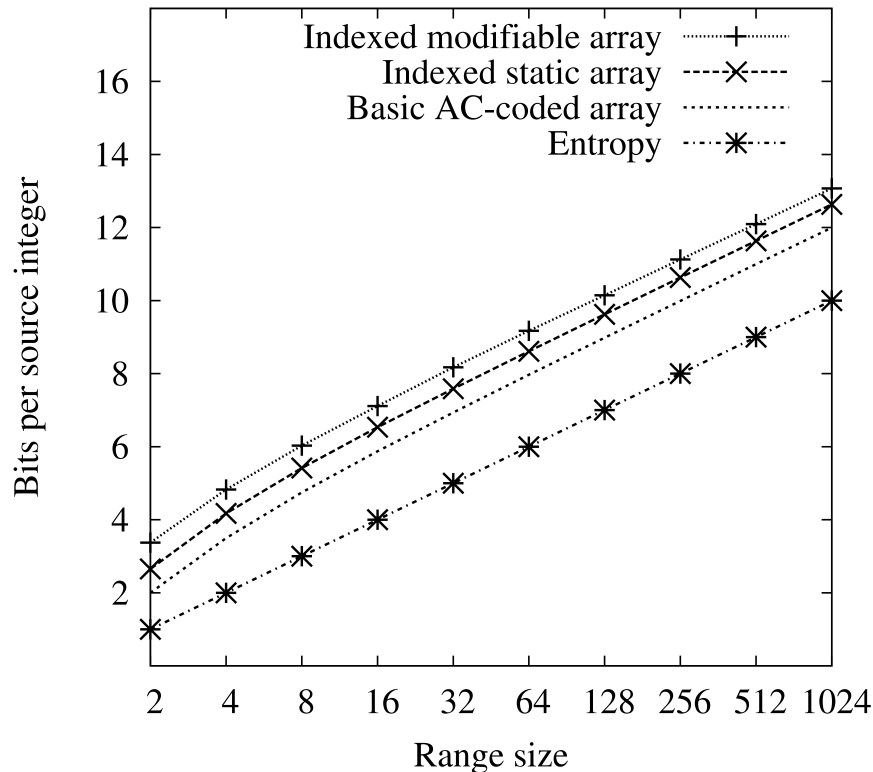
Compiler:

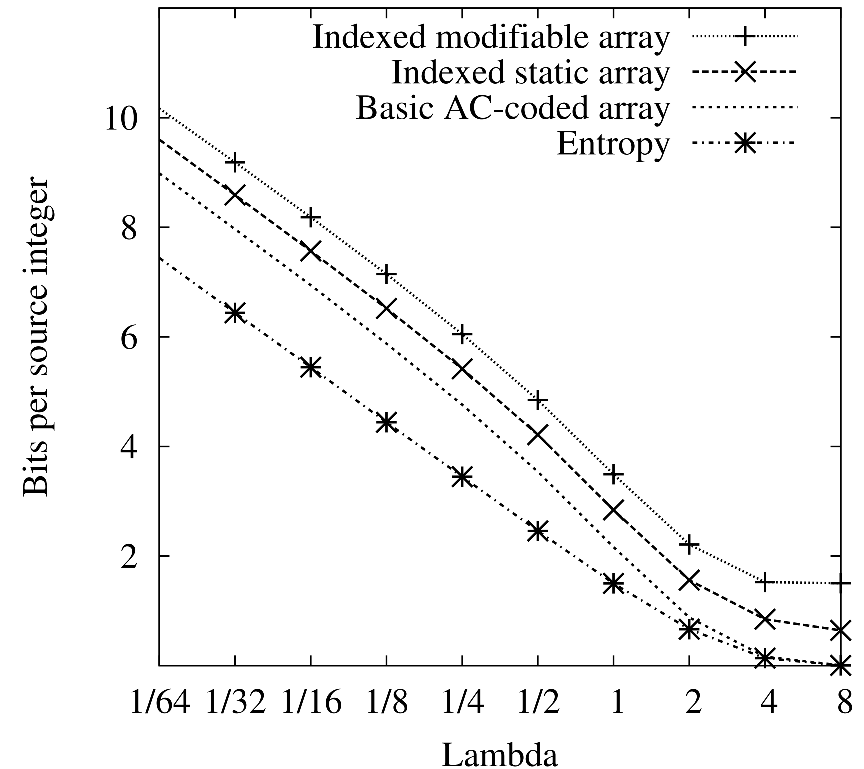
`gcc` with optimization `-O3`

Source code:

Available from Jukka's home page

Experimental results: Overhead

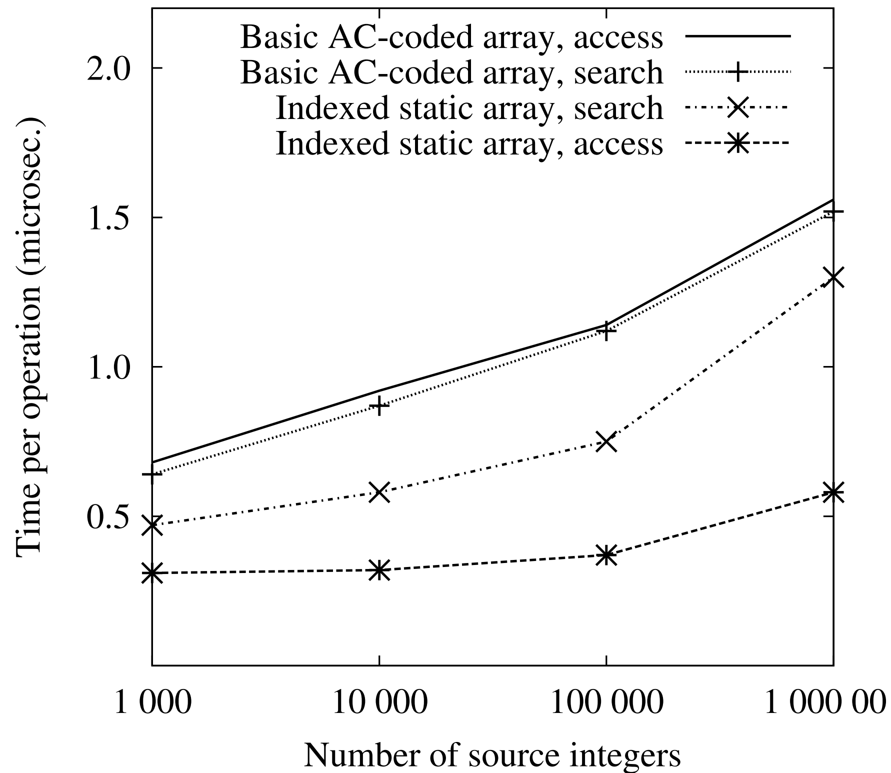


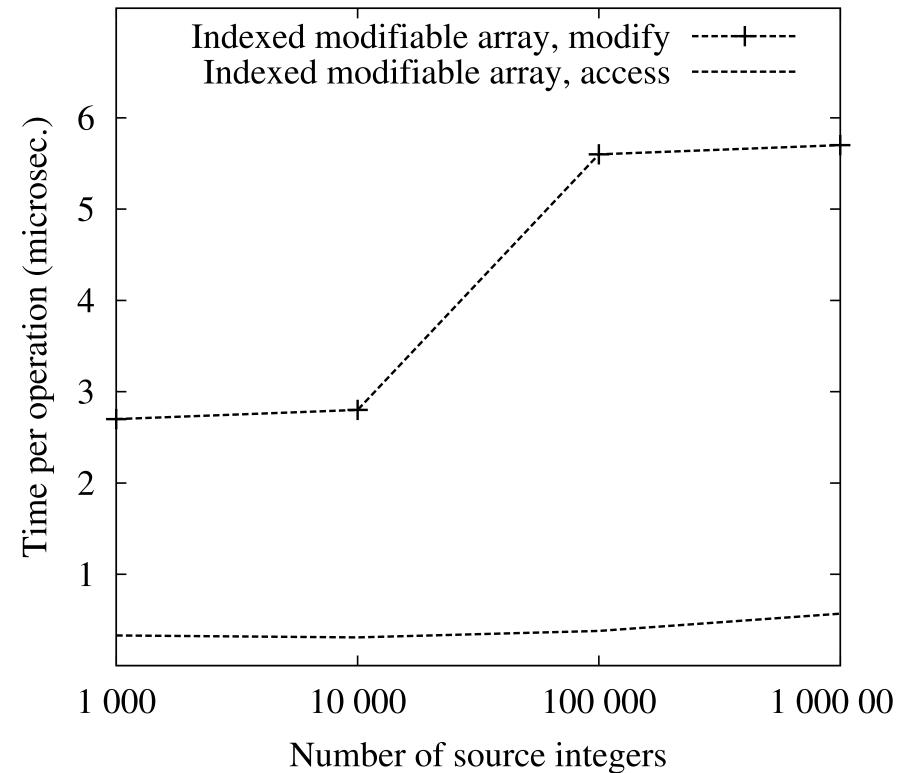


– entropy of x_i: expected information content of x_i

– for a random floating-point number y_i, $y_i \geq 0$, $x_i = \left\lfloor -\frac{\ln(1-y_i)}{\lambda} \right\rfloor$

Experimental results: *access*, *search*, *modify*





– uniformly-distributed integers drawn from [0..63]

Further work

Theory

- Try to understand better the trade-off between the speed of $access$ and the amount of overhead in the data-aware case.

Applications

- Can some of you convince me that compressed arrays are useful—or even necessary—in some information-retrieval application(s)?

Practice

- As to the speed of $access$, we showed that $O(\lg\lg(n+s))$ is better than $O(\lg(n+s))$. Can you show that $O(1)$ is better than $O(\lg\lg(n+s))$?
- Independent of the theoretical running time, can one get the efficiency of $access$ closer to that provided by uncompressed arrays?

To do

- A thorough experimental comparison!