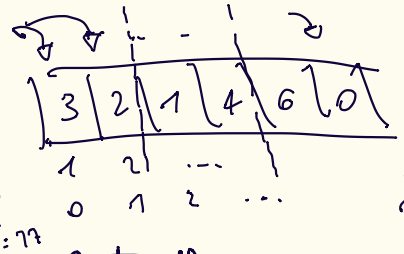
SUM OF n-integers:



- given $A = [1, n]$ => scan A and accumulate the sum in a temporary variable
  => $O(n)$

— > —

- Family of algorithms: $A_{s,b}$

$b = 2$ $\quad |A| = 6$ $\quad j = 0, \dots, \frac{6}{2} - 1$

$j = 0, 1, 2$ $\quad A = [3, 2, 1, 4, 6, 0]$ (b_0, b_1, b_2)

- A divided logically in blocks of b items:
  - b = # of items per block
  - Block => $A_j = A[j \cdot b + 1 ; (j+1) \cdot b]$ for $j = 0, 1, 2, \dots, \frac{n}{b} - 1$

  - $A_2 = A[2 \cdot 2 + 1 ; (2+1) \cdot 2] = A[5, 6]$

- Sum all items in a block before moving to the next one (that is s-blocks apart on the right)
- A is considered cyclic => when the next block lies out of A (starting)
  => algorithm wraps around it starting again from its beginning
- Not all value of s allows to keep account all A's blocks. (*)

(*) COPRIME: a, b are coprime iff: they have NOT common divisors, except 1, -1

=> but if s coprime (*) with $\frac{n}{b}$
then $[s \cdot i \bmod (\frac{n}{b})]$ generates a PERMUTATION $\{0, 1, \dots, \frac{n}{b} - 1\}$
thus $A_{s,b}$ touches all the block (=> sum of the integers)

SCAN BLOCK j

$j = i \cdot s \bmod \frac{n}{b}$

- DIFFERENT PATTERN MEMORY ACCESSES
  - Varing s, b:
    - sequencial scan: $s = b = 1$
    - block-wise access: large b
    - random-wise access: large s

- All algorithms $A_{s,b}$ are equivalent from a computational POV (read n integers => n steps)
- but pratically POV, different performances as array A grows:
  - for a growing n => data will be spread above Memory layers:
    (bandwith, latency, capacity, access methods)

• EVALUATING THE COST

- cpu time = n
- space occupancy = $\frac{n}{B}$ pages

1) Case $(s=1) \Rightarrow A_{1,b}$: scans A rightward by taking $\left(\frac{n}{B} I/Os\right)$
(independently from values of b.

2) Case ($s=2$ and b = some value $< B$)
↳ divides the block-size B

=> every block of size B, consist of $\frac{B}{b}$ logically small blocks

=> algorithm will visit just half of these logically small blocks
(cuz I'm jumping by $s=2$)

=> each B-sized page is half utilized in summing process

=> $\left(\frac{2n}{B} I/Os\right)$ (cost grows by 2 I'm jumping)

=> generalize the formula:

$$\min\left\{s, \frac{B}{b}\right\} \times \left(\frac{n}{B}\right) I/Os$$

=

MEMORY CONSIDERATIONS

$M \simeq$ GB of datas
→ data I'm processing are little bit more than M size

• Assume input size $\boxed{n = (1+\varepsilon)M}$ ($n = M + \varepsilon M$) is larger than internal memory size of factor $\varepsilon > 0$ =>

→ How much (ε) impact on the avg cost of an algorithm step, given that it may access to a data in internal memory or in the disk?

=> $p(\varepsilon)$ = "probability of an I/O fault" => probability of accessing the disk = $\frac{\varepsilon M}{n}$ (items out of the Memory / n = total # of items)

example: if $p(\varepsilon) = 1$ => algorithm always access its data on disk

if $p(\varepsilon) = \frac{\varepsilon}{1+\varepsilon}$ => algorithm has a full random behaviour

$$\frac{(\varepsilon)\cdot M}{(1+\varepsilon)\cdot M} = \frac{\varepsilon M}{(1+\varepsilon)M} = \frac{\varepsilon M}{n}$$

$\frac{\varepsilon M}{n} = \frac{\varepsilon M}{(1+\varepsilon)M} = \boxed{\frac{\varepsilon}{1+\varepsilon}}$

if $p(\varepsilon) = 0$ => working set of the algorithm fits in M, no I/Os
=> $\varepsilon = 0$ => no disk I/Os = $\frac{0}{1+0} = 0$ // no prob accen disk

=> EVALUATE: avg time of a step:

1) a = fractions of steps which are load or write (Memory or Disk)
2) $1-a$ = fractions of steps of computation (CPU)

=> avg time of a step: $(1-a) \cdot$ computation cost (suppose 1) $+ a \cdot$ cost of memory access (M or D)

$\Rightarrow (1-a)\cdot 1 + a\,[\text{access in M and access } 1-p(\varepsilon) + \text{access in disk and access } p(\varepsilon)]$

≤ 1 (1) ≤ 1 (C)

$= (1-a)\cdot 1 + a\,[1\cdot(1-p(\varepsilon)) + C\cdot p(\varepsilon)]$

prob. to go to D

- $(1-a)\cdot 1$: fraction of operations of computations; CPU step
- a: fraction of operations of memory
- $1\cdot(1-p(\varepsilon))$: memory step; go to M
- $C\cdot p(\varepsilon)$: I/O step; go to D

$\Rightarrow$ overall avg time of a step

$\Rightarrow \simeq a\cdot C\cdot p(\varepsilon)$ = $a \approx 30\%$, $10^5 \leq C \leq 10^6$

- a: dominated by; fraction of Memory
- C: fraction of IOs
- $p(\varepsilon)$: cost of an I/O; depends of size of the data

$\Rightarrow a\cdot C\cdot p(\varepsilon) = 30\%\cdot 10^6\, p(\varepsilon) = 0{,}3\cdot 10^6\cdot p(\varepsilon) =$

$= 300{.}000\cdot p(\varepsilon)$

$\Rightarrow$ Compared to CPU and Memory, this algo has a cost which is $300{.}000\cdot p(\varepsilon)$

if $p(\varepsilon)$ grows $\Rightarrow$ avg cost of a step will be in the order of 300.000

$p(\varepsilon)$ = go to Disk

$\Rightarrow$ 5 order magnitude more than internal memory or disk

also if over 1.000 accesses on Memory, 1 goes to the disk seems a very low amount

$p(\varepsilon) = \frac{1}{1000} \Rightarrow 300000\cdot \frac{1}{1000} = 300$ more slow!!!

Comparing 3 ALGOs:

$T(n) =$ 1°: n; 2°: n^2; 3°: 2^n;

- fix time t in which I want to run the algorithm:
  - which is the n such that $t(n) = t$?
    (find n which takes 10 minutes to run the algorithm)

1) $n = t$
2) $n^2 = t \Rightarrow n = \sqrt{t}$
3) $2^n = t \Rightarrow \log_2 2^n = \log_2 t \Rightarrow n = \log_2 t$

$\Rightarrow$ more polinomials grow $\Rightarrow$ less is amount of data I can process in a given time t $\Rightarrow$ INVERSE POLINOMIAL

- What happens if I have a k times more faster machine?
  (Instead of t, I have $t\cdot k$) $\Rightarrow k\cdot t$ time

1) $n = k\cdot t$ data processing $\Rightarrow$ 1° algo take full advantage of a k powerful machine
2) $n = \sqrt{k\cdot t} = \sqrt{k}\cdot\sqrt{t}$ $\Rightarrow$ amount of data can process, increased by a factor $\sqrt{k}$
3) $n = \log_2 k\cdot t = \log_2 k + \log_2 t$ ($\log_2 k$: very small)

BAD STUFF: increasing the size data I can process by an addition (not multiplication) and the addition is the logarithm of the improvement factor of the machine

$\Rightarrow$ need design polynomial algorithms that must be linear / sublinear

$T(n) = \log_2 n$

$T(n) = t$

$(t)^2 = (\log_2 n)^2$

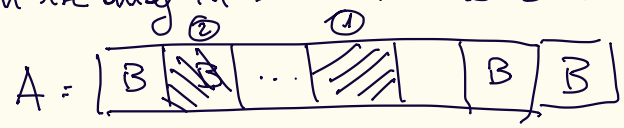
$n = t^2$ $\quad (\log_2 t)^2 = 2^2 = 4$

BINARY SEARCH

: pick item in the middle
- comparison
- go left/right

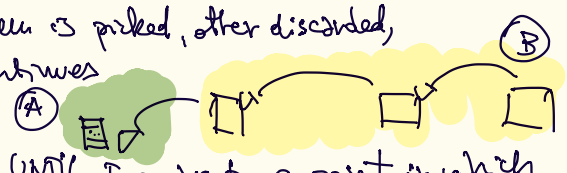
- Binary search: $\Theta(\log_2 n)$
  (BECAUSE IT'S OPTIMAL, you cannot do better!)

- I/O: logically partition the array in blocks of size B (on disk)



- pick the middle = is 1 item, the item is stored in a block, 1 block is picked in M
- then we jump (assume element is > where searching) => 1 I/O for 1 block, pick 1 item discard all |B|-1 items
  => fetch ① block, 1 item is picked, other discarded, algorithm continues

=> accessing items stored in different blocks



UNTIL I arrive to a point in which the range of the Binary search is < B

=> for some time you fetch blocks, at some time you stop to fetch blocks, even if BS continues
=> count both the phases

- In (A) I do $(\log_2 B)$ steps and 1 I/O. (everything is done in the same block)
- In (B) I do: $(\log_2 n)$ => binary search

last $\log_2 B$ are done inside 1 block

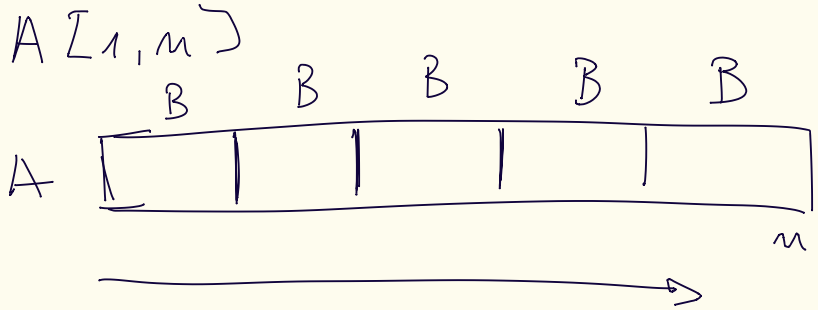
=> overall I do $\log_2 n - \log_2 B$ and all of them need 1 I/O for each BS step

1 I/O

=> COST: $\log_2\left(\frac{M}{B}\right)$ I/Os

if $n < B$ => array fits in a block => CONSTANT # I/Os

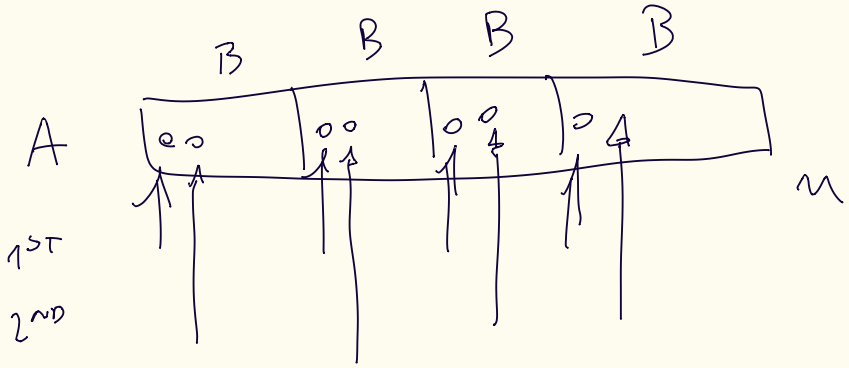
- Given array of integers, sum A



steps: n (RAM)

=> #I/Os = $O(\frac{n}{B})$ I/Os Level

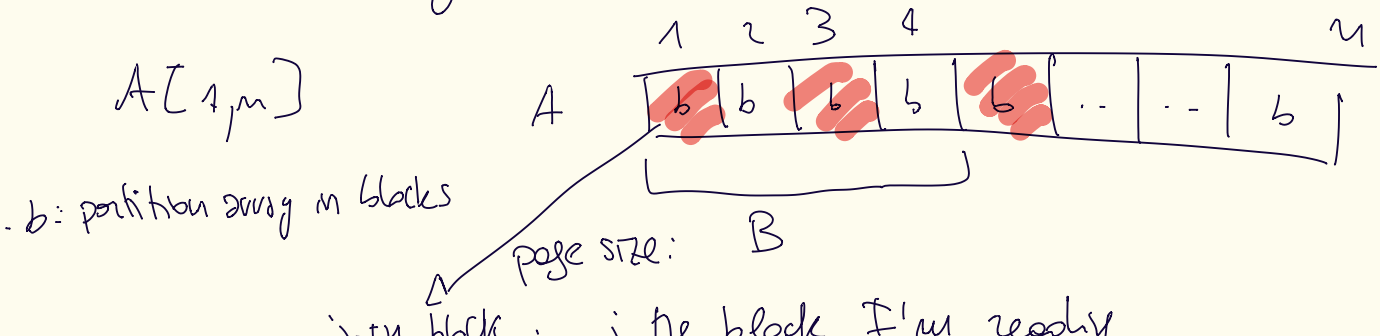
=> I can also



=> I can jump as sum the items in RAM still $O(n)$

- with jumps is $O(n)$ I/Os => worst if ≠ jump

- Create a family of algorithms: $A_{s,b}$



- b: partition array in blocks

$A_{s,b}$: - sums items in a block of size b
- then jump to the next s-block

jump block

-s = 2

∘ from left to right $A_{1,1}$

I/O complexity depends on parameters: s, b

∘ $j = i \cdot s \mod \frac{n}{B}$ where i goes from 1 to $\frac{n}{B}$

block number

this is a permutation when s and $\frac{n}{B}$ are coprime

s=2 we take 2, 4, 6 ...
s=3 3, 6, 9

modulo allows to wrap from the beginning
at page B I have to touch it s-times
(s=2 1st time 2nd, 4th
2nd time 1, 3

I/O complexity: $O\left(\frac{n}{B} \cdot \min\left\{s, \frac{b}{B}\right\}\right)$

→ larger is the jump
⇒ larger are the I/Os