# Amortized Analysis
# Fibonacci Heaps

thanks MIT slides
thanks "Amortized Analysis" by Rebecca Fiebrink
thanks Jay Aslam's notes

# Objectives

- Amortized Analysis
  - potential method
- Fibonacci Heaps
  - construction
  - operations

# running time analysis

- typical: Algorithm uses data-structure and operations
  - structures: table, array, hash, heap, list, stack
  - operations: insert, delete, search, min, max, push, pop
- measure running time by analyzing
  - the sequence of operations,
  - their frequency
  - each operation running time (computation cost)

# Running Time Analysis

- determine the c = costliest/longest iteration
  - usually an outer loop of n iterations
  - overall n* (longest cost per iteration) = n*c
- Thats not very accurate!
  - not all iterations have the longest cost
  - perhaps some average technique can work, but how to prove?
- "compensate" : show that for every costly iteration, there must be other "cheap" iterations

# Example : binary counter

| bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 | cost (bits changed) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | N/A |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 3 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 4 |

- each row is a binary representation of the counter
  - increasing by one
  - right side: cost = how many bits require changes
- naive running time to increment from 0 to n :
  - each row may cost up to O(log n)
  - n iterations/increments would be O(n*logn)

# Example : binary counter

| bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |

| cost (bits changed) |
|---|
| N/A |
| 1 |
| 2 |
| 1 |
| 3 |
| 1 |
| 2 |
| 1 |
| 4 |

- true cost for n iterations: 1+2+1+3+1+2+1+4+... = 2n = O(n)
- reason: the iteration cost very rarely is O(log n)
  - O(logn) means changing a good number of bits
  - for one iteration of cost O(logn), there must be many "cheap" iterations

# binary counter amortization

| bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |

- Aggregation method: consider all n iterations
  - bit 0 changes every iteration => cost n
  - bit 1 changes for half of iterations => cost n/2
  - bit 2 changes quarter of iterations => cost n/4
  - bit 3 changes 1/8 of iterations => cost n/8
  - ... etc
- total cost : add up the cost per bit
  - n + n/2 + n/4+ n/8 + ... = 2n
- Aggregation method impractical, only works on toy examples like this

# Amortized Analysis

- $c_i$ = true cost of i-th operation/iteration
- $\hat{c}_i$ = amortized cost of i-th operation/iteration
  - we have to come up with $d_i$
- the cumulative amortized cant be smaller than the true cumulative cost, up to any iteration k

$$\forall k : \sum_{i=1:k} c_i \leq \sum_{i=1:k} \hat{c}_i$$

# Accounting Method

- assign the di amortized cost
- if overcharge some operation (di>ci) use the excess as "prepaid credit",
- use the prepaid credit later for an expensive operation

# Potential method

- associate a potential function $\phi$ with datastructure T
  - $\phi(Ti)$ = "potential" (or risk for cost) associated with datastructure after i-th operation
  - typically a measure of complexity/risk/size of the datastructure
- require $\hat{c}_i \geq c_i + \phi(T_i) - \phi(T_{i-1})$ for all i
- $\hat{c}_i$ = amortized cost (up to us to define)
- ci = true cost for operation i
- $\phi$ = potential function
- Ti = datastructure after ith operation

# Accounting Method for binary counter

| bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 | true cost ($c_i$) | amortized cost $\hat{c}_i$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | N/A | N/A |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 |
| 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 |
| 0 | 0 | 0 | 1 | 0 | 0 | 3 | 2 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 2 |
| 0 | 0 | 0 | 1 | 1 | 0 | 2 | 2 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 |
| 0 | 0 | 1 | 0 | 0 | 0 | 4 | 2 |

$\sum c_i$ $\sum_{i=1:k} \hat{c}_i$

- assign amortized cost of di=2 for each operation
- verify the amortized condition $\forall k : \sum_{i=1:k} c_i \leq \sum_{i=1:k} \hat{c}_i$

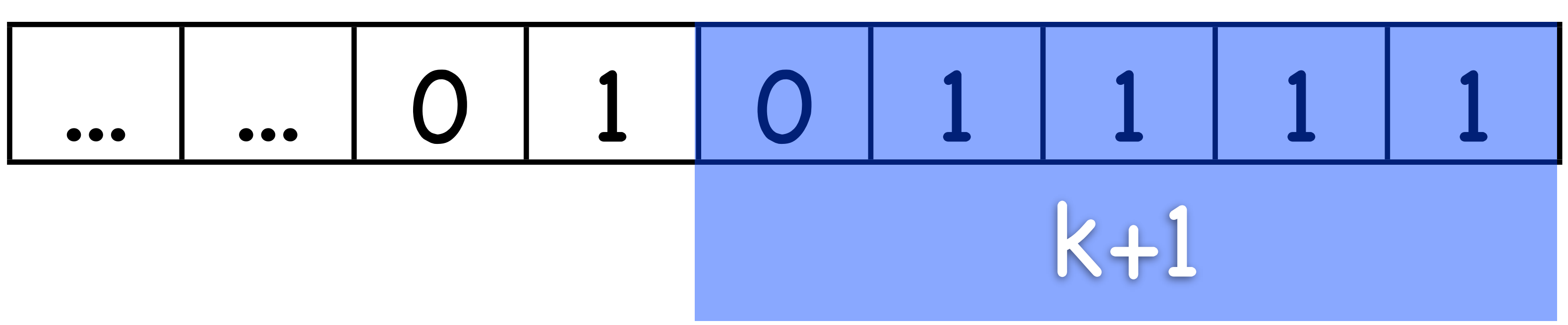
# Accounting Method for binary counter

| bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 | true cost ($c_i$) | amortized cost $\hat{c}_i$ | cum true cost $\sum c_i$ | cum amortized cost $\sum_{i=1:k} \hat{c}_i$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | N/A | N/A | N/A | N/A |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 1 | 2 |
| 0 | 0 | 0 | 0 | 1 | 0 | 2 | 2 | 3 | 4 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 4 | 6 |
| 0 | 0 | 0 | 1 | 0 | 0 | 3 | 2 | 7 | 8 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 2 | 8 | 10 |
| 0 | 0 | 0 | 1 | 1 | 0 | 2 | 2 | 10 | 12 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 11 | 14 |
| 0 | 0 | 1 | 0 | 0 | 0 | 4 | 2 | 15 | 16 |

- assign amortized cost of di=2 for each operation
- verify the amortized condition $$\forall k : \sum_{i=1:k} c_i \leq \sum_{i=1:k} \hat{c}_i$$

# Potential method for binary count

- define the potential $\phi(T_i)$ = the number of "1" bits
- verify $\hat{c}_i \geq c_i + \phi(T_i) - \phi(T_{i-1})$ for each operation
  - there is only one operation: "increment"
  - di=2 , amortized cost defined by us
  - before the operation i, at $T_{i-1}$, say there are k trailing 1 ones, before i-th increment
  - ci= true cost = k+1 bit changes: k of "1" bits made "0" (from right to left up to the first "0"); plus the first "0" made "1"
  - $\phi(T_i) - \phi(T_{i-1})$ = "1" gained - "1" lost = 1-k
  - equation becomes 2≥k+1 + 1-k, it checks out! di = 2 is good



# Stack operations - review

- stack is an array with LAST-IN-FIRST-OUT operations
- push(value) : put the new value on the stack (at the top)
- pop(n): take the top n values, return the, delete them from stack
- naive analysis for n operations : $n*O(n) = O(n^2)$
- better: for pop() to extract many elements, many push() must have happened before, each push is O(1)

| | | | | |
|---|---|---|---|---|
| | z | | | |
| c | c | | d | |
| b | b | b | b | b |
| a | a | a | a | a |
| | push(z) | pop(2) | push(d) | pop(1) |

# Accounting method for Stack

- account each push(x) with $2:
  - $1 for the actual push(x) operation, to add x to the stack
  - $1 credit for the possible later pop() operation that extracts x
- each pop(k) also $2, for any k
- so each operation is accounted with $2,
- total running time for n operations is 2*n = O(n)
- when pop(k) is called, each one of the popped elements have stored $1 to account for their extraction, O(k) time

# Potential method for Stack

- define the potential $\phi$(stack) = size(stack)
  - $\phi(T) = |T|$ ; T = the stack; $T_i$ = stack after i operations
- define the amortized costs: $d_{push}=2$ ; $d_{pop}=2$
- consider the true costs $c_{push}=1$ ; $c_{pop(k)}=k$
- prove that for each operation the potential satisfies the fundamental property (exercise)

$$\hat{c}_i \geq c_i + \phi(T_i) - \phi(T_{i-1})$$

- which means the amortized cost d=2 is valid.