Lecture 2: Wed Jan 08, 2003

today

- intro
  - sample alg'm: insertion sort
  - notation: psueudocode
  - model of computation: RAM
  - time analysis of insertion sort (using RAM)

getting started [CLRS Ch. 2]

- **algorithm?** a well-defined computational procedure (namely, a sequence of elementary computational steps) which takes an input value and produces an output value, according to a specified mathematical function.
- describing alg'ms: pseudocode
- pseudocode example

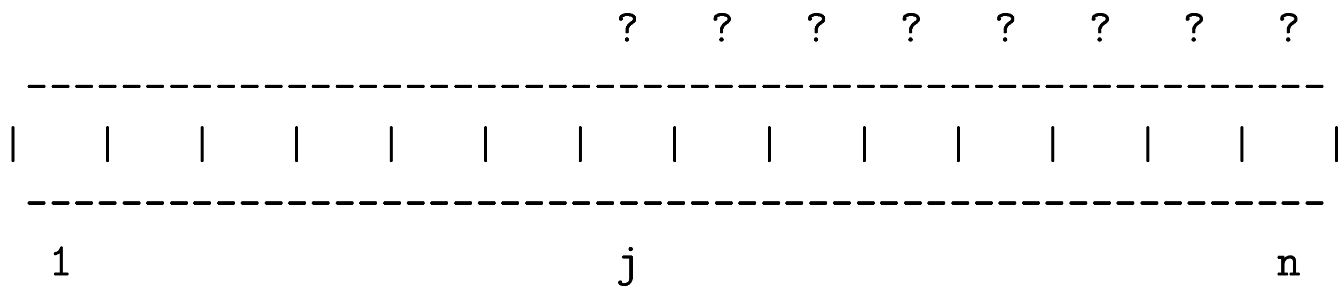
```
input:  integers a,b
output: a*b

sum <- 0
for j <- 1 to b do
  sum <- sum + a
return sum
```

- pseudocode conventions [CLRS p19]
  - indentation $\leftrightarrow$ block structure
  - while/for/repeat/if/then/else
  - loop counters retain values
  - ** or $\triangleright$ comment
  - variables local (unless stated otherwise)
  - parameters passed by value
  - boolean "short circuit" evaluation `j>0 AND A[j] < key`

problem: sorting
input: sequence of n numbers $(a_1, a_2, \ldots, a_n)$
output: permutation $(a'_1, a'_2, \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

one sorting algorithm: insertion sort

- idea   repeatedly insert $A[j]$ into sorted sublist $A[1 \ldots j-1]$
- insert how?
  - search for insert loc'n in sequence $A[j-1]$ $A[j-2]$ ...
  - move contents to right during search




```
InsertionSort(A)   **sorts A[1..n] in place  CLRS page 17
1  for j <- 2 to length[A] do
2    key <- A[j]
3    ** insert A[j] into sorted sequence A[1..j-1]
4    i <- j-1
5    while i>0 and A[i]>key do
6      A[i+1] <- A[i]
7      i <- i-1
8    A[i+1] <- key
```

insertion sort trace

```
      ??    ??    ??    ??    ??
53    21    47    62    14    38

            ??    ??    ??    ??
21    53    47    62    14    38

                  ??    ??    ??
21    47    53    62    14    38

                        ??    ??
21    47    53    62    14    38

                              ??
14    21    47    53    62    38


14    21    38    47    53    62
```

algorithm analysis

- resources required (time/space)
- correctness

resources required (time/space)

- how to measure?   implement and run (?)
- depends on input; language; machine; code; environment
- simplifying idea
  - select theoretical computer model
  - estimate resources on model

representation of non-negative integers

- let N = $\{0, 1, 2, \ldots\}$
- recall: for any b, n in N with $b > 0$,
  n has a unique base-b representation
- $13_{10} = 1111111111111_1$
- $13_{10} = 1101_2$
- $13_{10} = 111_3$
- $13_{10} = 31_4$
- $13_{10} = 23_5$
- $13_{10} = 21_6$
- $13_{10} = 16_7$
- $13_{10} = 15_8$
- $13_{10} = 14_9$
- $13_{10} = 13_{10}$
- $13_{10} = 12_{11}$
- $13_{10} = 11_{12}$
- $13_{10} = 10_{13}$
- most computers use a binary (base 2) rep'n

binary representation algorithm

```
precondition: n non-negative integer
postcondition: a is binary representation of n

m <- n      j <- -1
repeat
  j    <- j+1
  a[j] <- m mod 2
  m    <- m / 2
until m = 0
```

- trace: n=13

- correctness?
  - loop terminates — why? proof?
  - final value of j?
  - postcondition:

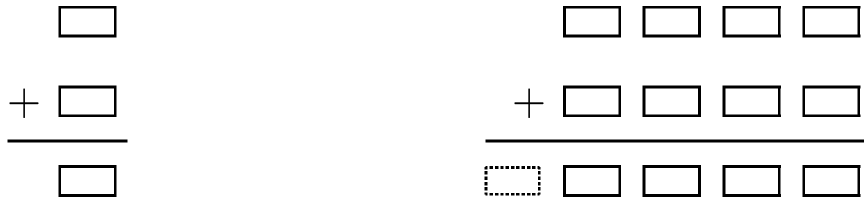
$$n = \sum_{t=0}^{j} a_t 2^t$$

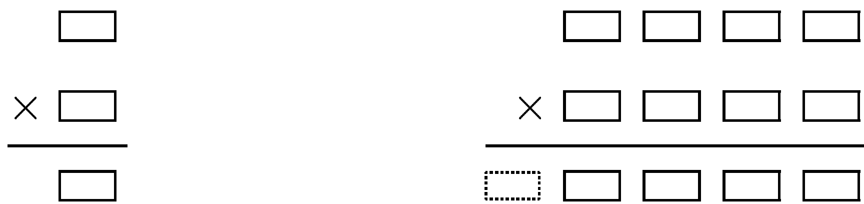
  - proof by induction? (see CS272 text)

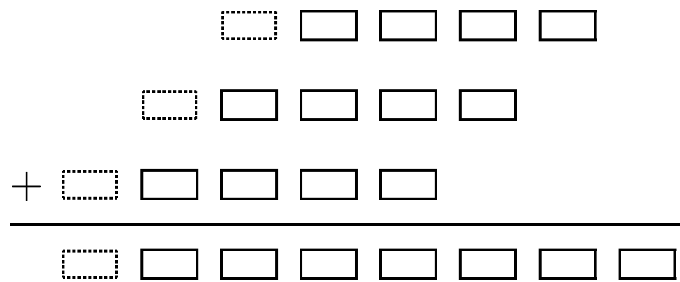
our model of computation: RAM

- RAM: random access machine [CLRS p21-22]
- every location indexed, can be accessed in constant (?) time
- usual variation: **uniform (unit) cost RAM model**
  - numbers not too large, so each number fits into one memory word
  - e.g. `x <- y` takes constant time
  - how long does $x + y$ take? $x * y$?
- more realistic variation: **log cost RAM**
  - numbers may be very large, and so not fit into one memory word
  - binary rep'n of number x needs about $\lg x$ bits
  - assume k bits per word of memory (e.g. 64)
  - basic operation on x takes time/space prop'l to number of words
  - e.g. `x <- y` takes prop'l to $(\lg \max\{x, y\})/k$ time
  - how long does $x + y$ take? $x * y$?
- unless otherwise stated, assume uniform cost RAM ☺

RAM arithmetic costs







	uniform	log*
add	1	$\approx (\lg n)/k$
mult	1	$\approx ((\lg n)/k)^2$

* assume k bits/cell

more details of typical RAM machine

- components
  - IT: input tape (read-only)
  - OT: ouput tape (write-only)
  - CU: computation unit (inc. program)
  - M: memory locations (each can store integer)
    M[0] M[1] M[2] ...
  - program: fixed user-defined instr. sequence
- properties
  - CU instructions (each usually via register/accumulator)
    - move data between memory
    - compare data and branch
    - binary arithmetic op'ns
    - read from IT to memory
    - write from memory to OT
- e.g. RAM instructions for $z \leftarrow x + y$?

```
M[0] := M[@x]  (*wherever x is*)
M[1] := M[@y]  (*wherever y is*)
add            (*M[0] := M[0] + M[1]*)
M[@z] := M[0]  (*wherever z is gets sum*)
```

resource analysis of RAM programs

- time: instructions executed
- space: memory locations used

RAM model time analysis

- model of computation: RAM
- problem: running time varies with input
- idea
  - estimate worst/average/best performance
  - make estimate a *function* of input *size*
  - e.g. sorting: size usually number of numbers
  - guarantee of performance ☺

when presenting analysis

for algorithm run times, remember to state what you have counted (e.g. RAM instructions? log RAM instructions? data moves? data comparisons? arithmetic operations? Pentium II clock cycles? etc.)

kinds of time analysis

- worst case
  - $T(n)$ max time over all inputs of size n
- average case
  - must specify input distribution over which average computed
  - most common: assume uniform (a.k.a. equiprobable) input distribution (all inputs of size n equally likely)
  - useful but usually difficult
- best case
  - useful for lower bounds
  - not otherwise useful: any alg'm can be modified to have fast best case
    (add: if input is [particular case] then return [particular answer])