Lecture 15

Algorithms & Data Structures

Goldsmiths Computing

February 11, 2019

Outline

Outline

Lecture

1. Pathfinding
2. Memoization
3. Dynamic programming

Lab

Graphs

- implement data structure
- implement minimum spanning tree
- implement shortest-pathfinding

(submission open Really Soon Now)

VLE activities

Graphs quiz

Statistics so far:

- A attempts: average mark B
- C students: average mark D
  - E under 4.00, F over 6.99, G at 10.00

Quiz closes at 16:00 on Friday 15th February

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Implicit data structures quiz

VLE activities (cont'd)

Implicit data structures quiz

VLE activities (cont'd)

Implicit data structures quiz

VLE activities (cont'd)

Binary heaps submission

Outline

Motivation

Random numbers needed for

- simulations
- games
- statistical software
- randomized algorithms

Definition

A random number is a number generated by some unpredictable process

- but: Laplace's demon

Definition

A random number is a number generated by some unpredictable process

- but: Laplace's demon

Pseudorandom Numbers

A pseudorandom number is a number generated by some process which is predictable and deterministic, but whose parameters are unknown
A pseudorandom number generator is an object which can generate a (long) sequence of pseudorandom numbers.

Operations

next! return the next random number from the generator (and update the generator's state)

seed![o] set the random number generator's state to something reproducible from the object o

Linear Congruential Generators

- single word of state, X
- generate the next pseudorandom number by computing $aX + c \bmod m$
- update the state to the new pseudorandom number

Example

$\mathrm{LCG}_{256}(29{,}35)$: $29X + 35 \bmod 256$

- 64, 99, 90, 85, 196, 87, 254, 233, 136, 139
- 93, 172, 159, 38, 113, 240, 83, 138, 197, 116
- 122, 245, 228, 247, 30, 137, 168, 43, 2, 93

Requirements

For full period of length m:

- m and c must be relatively prime
- $a - 1$ must be divisible by all prime factors of m
- $a - 1$ must be divisible by 4 if m is divisible by 4

(Hull-Dobel Theorem)

Problems with Linear Congruential Generators

- low period of some bits
  - *e.g.* in $29X + 35 \bmod 256$, sequence alternates odd/even
- serial correlations
  - choosing points in (2D-/3D-)space by generating successive random numbers severely restricts possibilities
- predictability
  - knowing m, can deduce a and c with only three successive random numbers

Take home message:

Do not use Linear Congruential Generators

- C `rand`
- C++ `minstd_rand`
- Java `java.util.Random`
- Javascript `Math.random`

(unless you know what you're doing)

Alternative random number generators

Mersenne Twister 19937

- period 2^{19937}-1; 19937 state bits
- (not cryptographically secure)
- (pathological zero states)

xorshift, xoroshiro

- period 2^{128}-1; up to 128 bits of state;
- fast, non-correlated outputs
- (not cryptographically secure)
- (lowest bit linear-feedback weakness)

ISAAC, arc4random

- based on RC4, cryptographically secure

Work

1. Reading
   - CLRS, chapter 5
   - TIFU by using Math.random()
   - Dual EC: A Standardized Back Door

Outline

Motivation

- sorting is a fundamental operation
- intermediate step in many other algorithms

Definition

Any kind of search algorithm using a total order relation to compare pairs of elements to decide which should precede the other.

input a sequence of objects $s_0 \ldots s_{N-1}$

output a reordering of the sequence such that $s'_0 \leq s'_1 \leq s'_2 \leq \ldots \leq s'_{N-1}$

Total order relations

transitivity if $a \leq b$ and $b \leq c$ then $a \leq c$

totality $a \leq b$ or $b \leq a$

Bogosort

Require: s :: sequence
  while ¬SORTED?(s) **do**
    PERMUTE(s)
  end while
  return s

Complexity analysis

Time complexity

- there are $N!$ permutations of a sequence of N elements
- in the worst case the sorted permutation will be the last one

$$\Longrightarrow \Omega(N!)$$

Insertion sort

To sort a sequence: repeatedly insert the next unsorted element into its correct place in the sorted sequence.
Properties:

- stable
- straightforward
- in-place for arrays
    - also adaptible for in-place sorting of linked lists

Insertion sort

```
function INSERTIONSORT(s)
    for 1 ≤ j < LENGTH(s) do
        key ← s[j]
        i ← j−1
        while i ≥ 0 ∧ s[i] > key do
            s[i+1] ← s[i]
            i ← i - 1
        end while
        s[i+1] ← key
    end for
end function
```

Complexity analysis

Time complexity

- $N - 1$ iterations;
- for iteration number j, worst-case j array writes

$$\implies \Theta(N^2)$$

Space complexity

Only constant space required for running function:

$$\implies \Theta(1)$$

Work

1. Reading
   - CLRS, sections 2.1, 2.2
2. Investigate other quadratic sorting algorithms, for example:
   - selection sort
   - bubble sort
   - odd-even sort.

   What advantages and disadvantages do they have relative to insertion sort?
3. Questions from CLRS

   2-2 Correctness of bubblesort

Merge (vector)

Require: a,b :: Vector
  function MERGE(a,b)
    al ← LENGTH(a); bl ← LENGTH(b); cl ← al + bl
    c ← **new** Vector(cl)
    ai ← bi ← ci ← 0
    while ci < cl **do**
      if ai = al **then**
        c[ci] ← b[bi]; bi ← bi + 1
      else if bi = bl ∨ a[ai] ≤ b[bi] **then**
        c[ci] ← a[ai]; ai ← ai + 1
      else
        c[ci] ← b[bi]; bi ← bi + 1
      end if
      ci ← ci + 1
    end while
    return c
  end function

Mergesort

```
function MERGESORT(s)
    sl ← LENGTH(s)
    if sl ≤ 1 then
        return s
    else
        mid ← ⌊sl/2⌋
        left ← MERGESORT(s[0...mid])
        right ← MERGESORT(s[mid...sl])
        return MERGE(left,right)
    end if
end function
```

Quicksort

To sort a sequence: choose a pivot element, and generate subsequences of elements smaller and larger than that pivot element; sort those subsequences, and combine with the pivot.

Properties:

- in-place sort
- no extra heap storage required (and low stack space requirement)
- (only works on arrays)

Quicksort

```
function PARTITION(s,low,high)
    pivot ← s[high-1]
    loc ← low
    for 0 ≤ j < high-1 do
        if s[j] ≤ pivot then
            SWAP(s[i],s[j])
            i ← i + 1
        end if
    end for
    SWAP(s[hi],s[i])
    return i
end function
```

Quicksort

```
function QUICKSORT(s,low,high)
    if low < high then
        p ← PARTITION(s,low,high)
        QUICKSORT(s,low,p)
        QUICKSORT(s,p+1,high)
    end if
end function
```

Complexity analysis

Time complexity: partition

- $N - 1$ iterations, each with (worst-case) one SWAP
- final SWAP at the loop epilogue

$$\implies \Theta(N)$$

Time complexity: quicksort

$$T(N) = T(N - p) + T(p - 1) + \Theta(N)$$

- depends on value of p!
- (we'll come back to this)

Complexity bounds

How efficient can comparison sorts be?

- how many possible permutations are there of a sequence of N distinct elements?
- how many of those possible permutations are sorted?
- how much information does a single comparison give?

Work

1. Reading
   - CLRS, section 2.3; CLRS, chapter 7
   - Jon Bentley, *Programming Pearls*, Column 11: sorting
2. Questions from CLRS

   Exercises 2.1-1, 2.1-2, 2.2-2, 2.3-1

Outline

Motivation

Random permutations are useful for many applications:

- games with chance
- work distribution across a computational cluster
- component of randomized algorithms

Definition

Shuffling is the operation of taking a linear collection of items, and returning the collection with the items reordered according to a (uniformly) random permutation.

Shuffling by sort, broken version

```
function RandomComparison(x,y)
    return random() − 0.5
end function
function BadShuffle1(A)
    return sort(A,RandomComparison)
end function
```

Shuffling by sort, better version

```
function AttachRandom(A,T)
    for 0 ≤ i < length(A) do
        lookup(T,A[i]) ⟵ random()
    end for
end function
function IndexedRandomComparison(x,y)
    return lookup(T,x) - lookup(T,y)
end function
function ShuffleBySort(A)
    T ⟵ new HashTable()
    AttachRandom(A,T)
    return sort(A,IndexedRandomComparison)
end function
```

Complexity

Space

hash table with N entries, plus whatever space SORT needs

$$\Longrightarrow \Omega(N)$$

Shuffling by swap, broken version

```
function BadShuffle2(A)
    N ← length(A)
    for 0 ≤ i < L do
        r ← random()
        j ← ⌊N × r⌋
        swap(A[i],A[j])
    end for
end function
```

Fisher-Yates shuffle

```
function FisherYates(A)
    for N > i > 0 do
        r ← random()
        j ← ⌊(i+1) × r⌋
        swap(A[i],A[j])
    end for
end function
```

Complexity

Space

Only temporary variable space needed

$$\Longrightarrow \Theta(1)$$

Time

- N−1 iterations;
- constant work at each iteration

$$\Longrightarrow \Theta(N)$$

Work

1. Find out why BadShuffle1 and BadShuffle2 are bad:
   - implement BadShuffle1 and BadShuffle2;
   - run them each 60000 times on a test input of `[1,2,3]`, and record how often each possible output comes up;
   - compare against how often each possible output *should* come up