Lecture 17

Algorithms & Data Structures

Goldsmiths Computing

March 4, 2019

Outline

Outline

Lecture

1. Number representations
   - fixed point
   - floating point
2. Numeric operations and algorithms
   - bitwise operations
   - arithmetic operations
   - optimizations for multiplication
   - population count

Lab

Random number generators

- Linear congruential
- Xorshift

VLE activities

Numbers 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 8th March

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

VLE activities (cont'd)

Random number generators submission

Outline

Motivation

- specialised sorting algorithm
- more information than just comparisons available
  - *e.g.* that items are keyed by small integers
  - recall sorting by age
- for when $O(N \log N)$ isn't good enough

Components

1. for each key, (efficiently) determine how many elements of the input are smaller than that element;
2. for each key, directly compute the position of that key in the sorted result;
3. for each element, place it in its final position.

Counting sort

```
function COUNTING-SORT(A,k)
    R ← new array(LENGTH(A))
    C ← new array(k)
    for 0 ≤ j < LENGTH(A) do
        C[A[j]] ← C[A[j]] + 1
    end for
    for 0 < i < k do
        C[i] ← C[i] + C[i-1]
    end for
    for LENGTH(A) > j ≥ 0 do
        R[C[A[j]]-1] ← A[j]
        C[A[j]] ← C[A[j]] − 1
    end for
    return R
end function
```

Complexity analysis

Space

- one temporary array C
- return value array R

$$\Longrightarrow \Theta(N + k)$$

Time

- iterate over input array A
- iterate over temporary array C
- iterate over return value array R

$$\Longrightarrow \Theta(N + k)$$

Work

1. Implement counting sort for arrays of integers between 0 and 100. How will you test your implementation?
2. Questions from CLRS

   Exercises 8.2-1, 8.2-4

   8-2 Sorting in place in linear time

Outline

Motivation

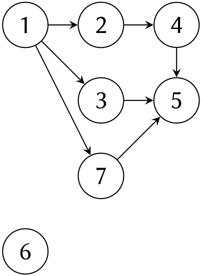
Given dependency information, generate a set of tasks in order so that dependent tasks are done after dependencies:

- spreadsheet recalculation
- Makefile target building
- database foreign key loading order
- serialization of data

Definition

A topological sort of a directed graph yields a linear collection of vertices such that if u and v are vertices and there is a edge from u to v, then u precedes v in the ordering.

Example



- 1, 2, 3, 7, 4, 5, 6
- 1, 6, 7, 2, 4, 3, 5
- 6, 1, 7, 3, 2, 4, 5

Kahn's topological sort

```
function KahnTS(G)
    L ← new DynamicArray(); S ← new Collection()
    for v ∈ vertices(G) ∧ ∄ e ∈ edges(G) : to(e) = v do
        insert(S,v)                          ▷ S: set of vertices with no incoming edges
    end for
    while ¬empty?(S) do
        v ← select!(S); push(L,v)                            ▷ add v to the end of L
        for e ∈ edges(G) ∧ from(e) = v do
            z ← to(e)
            remove-edge!(G,e)
            if ∄ f ∈ edges(G) : to(f) = z then
                insert(S,z)
            end if
        end for
    end while
    return L                          ▷ if G still has edges, then G was not a DAG
end function
```

Depth-first topological sort

```
function DFTS(G)
    L ← new List()
    UM ← new Set(VERTICES(G))
    TM ← new Set(); PM ← new Set()
    function VISIT(v)
        if v ∈ PM then
            return
        end if                          ▷ if v ∈ TM then we have found a cycle
        DELETE!(UM,v); INSERT(TM,v)
        for e ∈ EDGES(G) ∧ FROM(e) = v do
            VISIT(TO(e))
        end for
        DELETE!(TM,v); INSERT(PM,v)
        L ← CONS(v,L)
    end function
    while ∃ v ∈ UM do
        v ← SELECT!(UM)
        VISIT(v)
    end while
    return L
end function
```

Relation to relations

Consider a relation R such that R is irreflexive, antisymmetric and transitive (a strict partial order). A topological sort of the graph induced by that relation will convert the partial order into a total order.
The transitive closure of any directed acyclic graph corresponds to a strict partial order.

Work

1. Reading
   - CLRS, sections 22.3, 22.4
   - DPV, section 3.3
2. Exercises and problems
   - CLRS, exercises 22.3-2, 22.4-1, 22.4-5
   - DPV, exercises 3.3, 3.14

Outline

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

$$\Longrightarrow \Theta(N)$$

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

- generalization of maximum operation
- component of solving real problems:
    - return the ten best matches to a query
    - return the median of this set of data

Definition

Selection is the operation of selecting the k^{th} largest element (with respect to some order relation) from a dataset of N elements.

Maximum

```
function MAXIMUM(A)
    result ← −∞
    for 0 ≤ i < LENGTH(A) do
        if A[i] > result then
            result ← A[i]
        end if
    end for
    return result
end function
```

Maximum

```
function MAXIMUM(A)
    result ← −∞
    for 0 ≤ i < LENGTH(A) do
        if A[i] > result then
            result ← A[i]
        end if
    end for
    return result
end function
```

Complexity analysis

- time: $\Theta(N)$
- space: $\Theta(1)$

Second

```
function SECOND(A)
    max ← −∞; result ← −∞
    for 0 ≤ i < LENGTH(A) do
        if A[i] > result then
            if A[i] > max then
                result ← max
                max ← A[i]
            else
                result ← A[i]
            end if
        end if
    end for
    return result
end function
```

Second

```
function SECOND(A)
    max ← −∞; result ← −∞
    for 0 ≤ i < LENGTH(A) do
        if A[i] > result then
            if A[i] > max then
                result ← max
                max ← A[i]
            else
                result ← A[i]
            end if
        end if
    end for
    return result
end function
```

Complexity analysis

- time: $\Theta(N)$ (but twice as much as for maximum)
- space: $\Theta(1)$ (but twice as much as for maximum)

kth

```
function KTH(A,k)
    maxes ← new collection(k)
    for 0 ≤ i < LENGTH(A) do
        if A[i] > SMALLEST(maxes) then
            REMOVE-MIN(A[i])
            INSERT(A[i],maxes)
        end if
    end for
    return MIN(maxes)
end function
```

kth

```
function KTH(A,k)
    maxes ← new collection(k)
    for 0 ≤ i < LENGTH(A) do
        if A[i] > SMALLEST(maxes) then
            REMOVE-MIN(A[i])
            INSERT(A[i],maxes)
        end if
    end for
    return MIN(maxes)
end function
```

Complexity analysis

maxes Array (unsorted)

- REMOVE-MIN is $\Theta(k)$
- INSERT is $\Theta(1)$
- REMOVE-MIN called $\Theta(N)$ times

$\implies \Theta(Nk)$

kth

```
function KTH(A,k)
    maxes ← new collection(k)
    for 0 ≤ i < LENGTH(A) do
        if A[i] > SMALLEST(maxes) then
            REMOVE-MIN(A[i])
            INSERT(A[i],maxes)
        end if
    end for
    return MIN(maxes)
end function
```

Complexity analysis

maxes Array (sorted)

- REMOVE-MIN is $\Theta(1)$
- INSERT is $\Theta(k)$
- INSERT called $\Theta(N)$ times

$$\implies \Theta(Nk)$$

kth

```
function KTH(A,k)
    maxes ← new collection(k)
    for 0 ≤ i < LENGTH(A) do
        if A[i] > SMALLEST(maxes) then
            REMOVE-MIN(A[i])
            INSERT(A[i],maxes)
        end if
    end for
    return MIN(maxes)
end function
```

Complexity analysis

maxes min-heap

- REMOVE-MIN is $\Theta(\log k)$
- INSERT is $\Theta(\log k)$
- each called $\Theta(N)$ times

$$\implies \Theta(N \log k)$$

median

Selecting k$^{\text{th}}$ element takes $\Theta(N \log k)$ time

- selecting median ($\frac{N}{2}$th element) takes $\Theta(N \log N)$ time
- no better (asymptotically) than a full sort!

Can we do better?

- yes!
- quickselect, like partial quicksort
- compute the k$^{\text{th}}$ element in $\Theta(N)$ time (worst case)

Quickselect

```
function QUICKSELECT(S,low,high,k)
    if low = high then
        return S[low]
    else
        p ← PARTITION(S,low,high)
        if p = k then
            return S[k]
        else if k < p then
            return QUICKSELECT(S,low,p,k)
        else
            return QUICKSELECT(S,p+1,high,k)
        end if
    end if
end function
```

Median of medians

How to choose pivot for quickselect (and quicksort)?

- bad choice leads to $\Theta(N^2)$ (quadratic) performance

Guaranteed good choice of pivot for partitioning:

- break sequence into groups of 5
- compute the median of each group
- compute the median of the medians and use that as pivot

Recurrence relation

$$T(N) \leq T\left(\frac{N}{5}\right) + T\left(\frac{7N}{10}\right) + \Theta(N)$$

Can show by strong induction (or Akra-Bazzi method) that

$$T(N) \in \Theta(N)$$

Work

1. Reading:
   - CLRS, sections 9.1, 9.2
2. Questions from CLRS:

   9.1 Largest i numbers in sorted order