Fundamental Algorithms
Lecture #4

Cluj-Napoca

CS, UTCN

Agenda

- Sorting – lessons learned
- Sorting in linear time
- Radix Sort
- Sorting – Closing Evaluation
- Elementary DS
  - Stacks and Qs
  - Lists

Sorting – lessons learned

- No direct method is optimal
- Yet, some of them are worth to be used in specific conditions. Which ones, when? Discussion.
- Stability is a desired property; not all strategies own it. Which do? Which not? Discussion.
- Advanced strategies (heapsort and quicksort) are optimal. However, it does not worth using them always. When not? Why? Discussion.
- **Cases** depend on the strategy (**algorithm**) **AND implementation**!
  - **Cases** are **not** fixed on the **problem**!!!
  - One best case of one solution might be worst case of another's

MergeSort

- Relies on merging 2 ordered arrays ($O(n)$)
- Divide et impera strategy
- Opposite to QuickSort:
  - divides fast = find middle $O(1)$
  - combines = merge $O(n)$
- By design always the best case: splits the data into 2 equal parts.
- $t(n)=2t(n/2) + O(n) \Rightarrow O(n\lg n)$
- Is it optimal? Why?
- How much additional space does it need?

QuickSort vs MergeSort

- Compare and contrast analysis
- Both sorting algorithms with divide et impera strategy

	QS	**MS**
Relies on:	divide (*partition*)	combine (*merge*)
Has default:	combine (NoOp)	divide (*middle index*)
Non recursive time:	O(n)	O(n)
Space:	in situ	needs additional space O(n)
Complexity:	O(nlgn) randomized	O(nlgn) always
When to use:	(very) large data/hybrid	very large data (external)

Sorting in linear time

- O(n) ? How? Isn't contradicting the lower bound, as the sorting problem has $\Omega(n\lg n)$?
- Counting Sort – additional **constraints** + **space**
- Each of the input elements is an int in range 1..k
- Idea:
  - $\forall x \in$ Input, **evaluate** (=count) the nb. of els. $\leq x$, $\mathbf{i_x}$
  - **Use $\mathbf{i_x}$ as an index** to place x in the Output, Out[i_x]<-x
  - Input/Output! Is **not** in-situ sort
- Ex: Input A[1..n]={2,7,3,1,2,9,2,...}
  - There are 5 elements $\leq$ 3 (1 vals of 1, 3 vals of 2, and itself)
  - So, Output B[5]<-3

Counting Sort

- All previous solutions are comparison-based
- A, B i/o arrays (O(n) space)
- C a counting array (O(k) space)
  - C[1..k], 1-k the range of els from input
  - C[i] counts the nb. of els from the input having the value ≤i
  - C is used as an index, to move the i^{th} el from input (i.e. take A[i]) to output (i.e. place in B[C[A[i]]])
- The Algorithm:
  - Evaluate C
  - Use C to move data

Counting Sort - code

```
CountingSort(A,B,k)
for i<-1 to k
   do C[i]<-0                      //initialize C

for j<-1 to length[A]
   do C[A[j]]<-C[A[j]]+1           //A's value acts as an index; all
                                   // A's vals increment the corresponding C
                                   //after the loop C[j]=nb of els =j
for j<-2 to k
   do C[j]<-C[j]+C[j-1]            // C[j]=nb of els ≤j
for j<- length[A] downto 1
   do B[C[A[j]]<-A[j]
      C[A[j]]<-C[A[j]]-1
```

Counting Sort – execution

```
CountingSort(A,B,k)
for i<-1 to k
   do C[i]<-0
```

A	1	2	3	5	3	2	1	Vals at input
B								Vals at output
C	0	0	0	0	0	NA	NA	Counter

```
for j<-1 to length[A]
   do C[A[j]]<-C[A[j]]+1
```

j	1	2	3	4	5
C	2	2	2	0	1

//the sequence counts how many els
//of each value are in the table

Trace step#2

A	1	2	3	5	3	2	1

j=1

j	1	2	3	4	5
C	1	0	0	0	0

j=2

j	1	2	3	4	5
C	1	1	0	0	0

j=3

j	1	2	3	4	5
C	1	1	1	0	0

j=4

j	1	2	3	4	5
C	1	1	1	0	1

j=5

j	1	2	3	4	5
C	1	1	2	0	1

j=6

j	1	2	3	4	5
C	1	2	2	0	0

Counting Sort – execution

```
CountingSort(A,B,k)
for i<-1 to k
   do C[i]<-0
```

A	1	2	3	5	3	2	1	Vals at input
B								Vals at output
C	0	0	0	0	0	NA	NA	Counter

```
for j<-1 to length[A]
   do C[A[j]]<-C[A[j]]+1
```

j	1	2	3	4	5
C	2	2	2	0	1

//the sequence counts how many els of
//each value are in the table

Counting Sort – execution – cont.

```
for j<-2 to k            //counts nb of els<=each value
  do C[j]<-C[j]+C[j-1]
```

j=2 (how many els <=2?)

j	1	2	3	4	5
C	2	4	2	0	1

j=3

j	1	2	3	4	5
C	2	4	6	0	1

j=4

j	1	2	3	4	5
C	2	4	6	6	1

j=5

j	1	2	3	4	5
C	2	4	6	6	7

Obs: There are 7 els <= 5; 6 els <=4; also 6 els <=3; (=> no element with value 4); …

Counting Sort – execution – cont.

```
for j<- length[A] downto 1
   do B[C[A[j]]<-A[j]
      C[A[j]]<-C[A[j]]-1
```

j=7 B[2]<-A[7]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2					

index	1	2	3	4	5
C	2	4	6	6	7

C[1]<-C[1]-1

index	1	2	3	4	5
C	1	4	6	6	7

j=6 B[4]<-A[6]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2		2_2			

index	1	2	3	4	5
C	1	4	6	6	7

C[2]<-C[2]-1

index	1	2	3	4	5
C	1	3	6	6	7

Counting Sort – execution – cont.

```
for j<- length[A] downto 1
  do B[C[A[j]]<-A[j]
     C[A[j]]<-C[A[j]]-1
```

j=5 B[6]<-A[5]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2		2_2		3_2	

index	1	2	3	4	5
C	1	3	6	6	7

C[3]<-C[3]-1

index	1	2	3	4	5
C	1	3	5	6	7

j=4 B[7]<-A[4]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2		2_2		3_2	5

index	1	2	3	4	5
C	1	3	5	6	7

C[5]<-C[5]-1

index	1	2	3	4	5
C	1	3	5	6	6

Counting Sort – execution – cont.

```
for j<- length[A] downto 1
   do B[C[A[j]]<-A[j]
      C[A[j]]<-C[A[j]]-1
```

j=3 B[5]<-A[3]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2		2_2	3_1	3_2	5

index	1	2	3	4	5
C	1	3	5	6	6

C[3]<-C[3]-1

index	1	2	3	4	5
C	1	3	4	6	6

j=2 B[3]<-A[2]

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B		1_2	2_1	2_2	3_1	3_2	5

index	1	2	3	4	5
C	1	3	4	6	6

C[2]<-C[2]-1

index	1	2	3	4	5
C	1	2	4	6	6

Counting Sort – execution – cont.

```
for j<- length[A] downto 1
   do B[C[A[j]]<-A[j]
      C[A[j]]<-C[A[j]]-1
```

j=1 B[1]<-A[1] C[1]<-C[1]-1

j	1	2	3	4	5	6	7
A	1	2	3	5	3	2	1
B	1_1	1_2	2_1	2_2	3_1	3_2	5

index	1	2	3	4	5
C	1	2	4	6	6

index	1	2	3	4	5
C	0	2	4	6	6

Counting Sort **is stable** (preserves in the output the relative input order between equal elements)

Which of the sorting algs are stable and which are not? Homework.

Counting Sort - eval

```
for i<-1 to k
  do C[i]<-0                               O(k)
for j<-1 to length[A]
  do C[A[j]]<-C[A[j]]+1                    O(n)
for j<-2 to k
  do C[j]<-C[j]+C[j-1]                     O(k)
for j<- length[A] downto 1
  do B[C[A[j]]<-A[j]
     C[A[j]]<-C[A[j]]-1                    O(n)
```

Counting Sort – eval –cont.

- $O(n)<\Omega(n\lg n)$ How?
- Does not rely on comparisons between the elements in the array! (elems are used as indices for the counting array)
- It's stable
- Looking forward for the parallel implementation

Radix Sort

- Card-sorting machine (Herman Hollerith, 1887)
- A strategy, rather than an “Algorithm”:
  - Examine the “under sorting” column
  - Distribute it into the corresponding bin
  - Bins are ordered (bin with 0’s before bin with 1’s aso)
  - Continue with the next column
- Order of examining cols: MSB vs LSB?
  - Both available
  - Homework: pros&cons for each method
- What sorting method used for sorting 1 col
  - A **stable** method (mandatory; otherwise LSB fails)
  - Either a direct stable or CountingSort (works very well as k=10)

Radix Sort – ex (LSB)

	V	V	V
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort – ex (MSB)

	V	V
329	329	329
457	355	355
657	457	436
839	436	457
436	657	657
720	720	720
355	839	839

Sorting by least significant digit (1s place) is not needed (why?)

Major drawback (which one?) Homework!

Radix Sort - evaluation

- Counting Sort the auxiliary sort (O(n+k))
- It is appropriate? Why?
- Needs d passes through Counting Sort (d=nb of bits in the n numbers) so O(dn+dk)
- If d=ct and k=O(n) => O(n) linear time

Sorting – Final Evaluation

- **Ω(nlgn)**
- None of the direct methods is optimal
- Stability is an important property (it is the implementation stable/unstable/undecidable, and not the strategies)
- ShellSort:
  - improves InsertSort (best direct strategy from various perspectives) by splitting the array into clusters (clusters are distance-based between the elements of the data, denoted as gaps)
  - apply InsertSort on clusters (Rationale: move elements further away from the original position, not just 1 position to the left);
  - changes gaps until gap=1
- HeapSort - optimal
  - Reason: it "remembers" comparisons done in previous steps keeping partial order structures
  - Resembles bubbleSort on subsets (branches); but uses a selection-based strategy
  - Used for priority queues

Sorting – Evaluation

Check:

http://cg.scs.carleton.ca/~morin/misc/sortalg/

visualizations of some comparison based sorting algorithms

Elementary DS

- Queues = set of data stored and accessed based on access policies
- Stacks and Queues = specific access policies
- **Stack**: LastInFirstOut **LIFO**
- **Queues**: FirstInFirstOut **FIFO**
- Implementations:
  - Array based
  - List based

Elementary DS

- All DS have the same basic operations
  - Add (insert)
  - Remove (delete)
  - Search
  - Update
  - Traverse
- All the rest are just combinations of the basic ones
- Important to know how they are handling the specific data and associated complexity

Stacks (with arrays)

- S[1..n]
- Access to the **first** element **only** (**top** el)
- LIFO policy
- Actions:
  - **Push** (= add/insert)
  - **Pop** (= extract/remove/delete)
  - **Stack-Empty/Stack-Full** (if size is associated – check for availability)

Stacks-code

Stack-Empty(S) //O(1)

```
if top[S]=0
   then return true
   else  return false
```

Push(S,x) //O(1)

```
top[S]<-top[S]+1        // top indicates the last occupied slot
S[top[S]] <-x           // does not check stack full (Homework)
```

Pop(S,x) //O(1)

```
if Stack-Empty(S)
   then error mess. "stack underflow"
   else top[S]<-top[S]-1
        return S[top[S]+1]
```

Queues (with arrays)

- Q[1..n]
- Access to the **first** element (***head***) on **reading**
- Access to the **last** element (***tail***) on **writing**
- FIFO policy
- Actions:
  - **EnQ** (= add/insert)
  - **DeQ** (= extract/remove/delete)
  - **Queue-Empty/Queue-Full** (Homework)

Queues-code

- Implementation **as a circular Q**
- Circular = no end; after Q[n] comes Q[1]

```
EnQ(Q,x)                                //O(1)
Q[tail[Q]]<-x     // tail indicates the first unoccupied slot
if tail[Q]=length[Q]
  then tail[Q]<-1
  else tail[Q]<- tail[Q]+1
```

- **Any possible error?**
- **No overflow test (the tail "eats" the head! Homework – fix it!)**

Queues-code-cont.

DeQ(Q,x) //O(1)

```
x <-Q[head[Q]]
if head[Q]=length[Q]
  then head[Q]<-1
  else head[Q]<- head[Q]+1
```

- **Any possible error?**
- **No underflow test (the head "reaches" the tail! Homework – fix it)**

Linked lists

- Dynamic DS
- Organized as:
  - Simple
  - Double
  - Circular
- Mandatory elements
  - key //+ the actual info; we skip it for now
  - next //pointer to the next el in list
  - previous //pointer to the prev in list ONLY if doubly linked list
- Particular cases:
  - prev[x]=nil in case x=head
  - next[x]=nil in case x=tail //ONLY for doubly linked list

Doubly linked lists - search

```
List-Search(L,k)            //O(n)
x<-head[L]
while x<>nil and key[x]<>k
  x<-next[x]
return x
```

Meaning:

When the returned is nil, means not found

When not nil, `x` points the actual searched (and found) element

Hw: rewrite as a recursive implementation. Time? Advantage? Disadvantage?

Doubly linked lists – insert

```
List-Insert(L,x)      //in the head; O(1)
//the el is already allocated and pointed by x;
next[x]<-head[L]
if head[L]<>nil      //Q was not empty before insert
  then prev[head[L]]<-x
head[L]<-x
prev[x]<-nil
```

Hw: insert in a certain position. **Steps**: Search for the position + link the element (4 pointers updates – 2 updates + 2 set)

Doubly linked lists – delete

`List-Delete(L,x)` //O(1)

//x is to be removed, and it **was found** by **`List-Search`**

```
if prev[x]<>nil                    //not the head of the list
   then next[prev[x]]<-next[x]
   else head[L]=next[x]
if next[x]<>nil                    //not the tail of the list
   then prev[next[x]]<-prev[x]
   else tail[x]=prev[x]
```

Any issues?

Dispose memory!!!

Sentinels

- Avoid testing for special cases (beginning/end of the structure)
- Each element is treated in an uniform manner
- Make the code easier to read and more efficient
- Sentinel=dummy el to which points prev[head] and next[tail]
- Transforms a doubly linked list into a circular list
- Qs and Stacks implemented with DLL with sentinels (Homework)

Lists implementation
Array vs Linked Lists

- Compare and contrast analysis

		Array	**Linked**
DS:		static	dynamic
Access:		direct (index based)	sequential (via traversal)
Complexity:			
Ins:	**at end**	O(1)	O(1)
	inner	O(n)	O(1) **(except for search)**
Del:	**at end**	O(1)	O(1)
	inner	O(n)	O(1) **(except for search)**
Space:		just data	data + pointers