Fundamental Algorithms
Lecture #3

Cluj-Napoca 16.10.2019

Rodica Potolea, CS, UTCN

Agenda

- Master Theorem – to be remembered
- Features to evaluate – review
- Heap structure - review
- QuickSort
- Selection
- QuickSort -updated

Master Theorem to remember/to keep close

a = number of recursive calls

b = division factor = ratio between original size over recursive size

c = degree of polynomial of the execution time of the sequence outside recursive calls: $f(n)= n^c$

- $$t(n) = \begin{cases} t_0 & \text{if } n<n_0 \\ at(n/b)+f(n) & \text{if } n>=n_0 \end{cases}$$

1. $q<1;\ a<b^c \quad => \quad O(n^c)$
2. $q=1;\ a=b^c \quad => \quad O(n^c * \log_b n)$
3. $q>1;\ a>b^c \quad => \quad O(n^{\log_b a})$

Features to evaluate - review

- Correctness
  - Partial and total
- Efficiency vs. optimality
  - Cases – what do they depend on
    - The problem to be solved
    - The algorithm solving the problem
    - The implementation of the algorithm
- Stability:
  - Stable vs unstable algorithm
- Determinism:
  - Deterministic vs nondeterministic behavior

Heap – as a data structure

- Static data structure (an array)
- Heap utilization when its size changes
- Heap_size – a data field
- Operations:
  - pop_heap extract the top from the heap
  - push_heap add one item to the heap

Heap – as a data structure – cont.

- pop_heap Extracts the top element O(1)
  - To restore the heap property (after the pop_heap):
    - Move bottom (last) element on top
    - Decrements the heap size
    - Heapify the whole (from 1 to the new size), to update the heap structure => O(lgn) time to RESTORE the heap property
  - If followed by a push, differently:
    - Just push on top & heapify (still O(lgn))
- push_heap
  - Increase the heap_size
  - Adds a new element at the bottom (last position in array)
  - Rebuild heap:
    - a bottom-up approach (bubble the bottom element upper in the heap, until it finds a larger-value parent) => O(h)=O(lgn)
  - If followed after a pop, differently:
    - Push on top (empty) & heapify (still O(lgn))

Heap – as a data structure – cont.

- build_heap
  - Repeats push_heap procedure
  - It takes $1+2\cdot1+4\cdot2+\ldots+n/2\cdot\lg n=O(n\lg n)$
- heap_sort
  - Build the heap (build_heap takes O(nlgn))
  - pop_heap (takes O(lgn))
  - add the poped element at bottom+1 (i.e. out of the heap, in the array)
  - It takes O(nlgn) (to build the heap)+ O(nlgn) (n times a pop operation)

Heap – comparison in building the heap

Approach	**Sol 1** (`heapify`)	**Sol2**(`pop/push`)
1 el approach	sinks the top (root)	bubbles a leaf
	O(h)	O(h)
all els(`build_heap`) approach	bottom-up (starts with the last nonleaf el)	top-down (adds a new leaf)
Time to build	O(n)	O(nlgn)
advantage	faster	variable dim
drawback	fixed dim	slower
usage	sorting	priority queues

Sorting – optimal strategies

- Optimal sorting = algorithm to sort in place (constant additional space) in O(nlgn) time
- In practice, quicksort, even not optimal (the original solution), behaves better than heapsort
- A good implementation of quicksort (by injecting various enhancements – see later) IS optimal

QuickSort

```
QuickSort(A,p,r)        //p, r -index of first, last el in
                        //the array A to order
if p<r                  //if proper array (=nonempty)
   then   q<-partition(A,p,r)    //q index returned
                  // at the boundary of the 2 partitions
          QuickSort(A,p,q)
          QuickSort(A,q+1,r)
```

t(n): T Master f(n)=n => c=1 (partition, next slide)

a=2 (2 rec calls)

b=?

Partition (as Hoare originally proposed the algorithm; in the original textbook – first edition)

```
Partition(A,p,r)          //p, r -index of the first, last el in the array
x<-A[p]  i<-p-1   j<-r+1   //pivot is the first element in the array
while i<=j do              //as long as left index to the left of right index
   begin
        repeat   j<-j-1
        until    A[j]<=x   //stop at the first smaller or equal element to pivot
        repeat   i<-i+1
        until    A[i]>=x   //stop at the first greater or equal element to pivot
        if i<j
                 then swap (A[i],A[j]) i<-i+1 j<-j-1
                 else return j
   end
```

Qs: (individual analysis! Hw!)

- the repeat-until loops stop on equal elements and swaps them. Why?
- the indexes i and j never go beyond the array boundaries. Why?
- First element pivot has an undesired worst case (leads $O(n^2)$ quicksort). Which is it? Why is it undesired?
- using A[p] as pivot is essential in this implementation. Why? Homework!
- using A[r] as pivot would cause error execution. Which one? Why? How can be avoided? Homework!

Partition (Hoare's update)

```
Partition(A,p,r)                    //p, r -index of first, last el in the array
x<-A[(p+r)/2]                       //or p, or r; doesn't matter
i<-p
j<-r
repeat
        while A[i]<x do i=i+1
        while A[j]>x do j=j-1
        if i<=j
        then    begin   swap(A[i],A[j])
                        i=i+1 j=j-1
                end
until (j<i)
```

Qs:

- Symmetric method. Works the same, whatever (middle, first, last) pivot is chosen.
- the while loops stop on equal elements and swap them. Why? Why not allowing them in the partition they already belong and change the loops conditions to nonstrict inequalities?
- In case i=j elements are swapped. It is redundant! Why to swap them? So can we change `if i<=j` into `if i<j`? Any trap?

QuickSort – eval.

b=? It depends on the case.

Cases DEPEND on the pivot choice, hence on the implementation!

Best: each partition divides the array into 2 equal parts => b=2 (in the Master theorem)=> O(nlgn)

Average: it can be shown it is close to the best case

Worst: each partition divides the array into arrays containing 1 element only and the rest of the elements => rec. calls each time on (1) and (n-1) elements respectively => $n+(n-1)+(n-2)+\ldots = O(n^2)$

(for the first/last element chosen as pivot, ordered array is the worst case!!!!) TO BE AVOIDED!

QuickSort – eval.

- Not an optimal alg $O(n^2) > \Omega(n \cdot \lg n)$
- $O(n\lg n)$ for best and average case
- Worst case occurs seldom
  - How seldom?
- Property of data to enter worst case?
  - How does it depend on the implementation?
  - What factor(s) impact the case?
  - pivot (for Partition) first element worst case?
  - pivot middle element worst case?
- How can ensure we **NEVER** enter the worst case?
- Always enter the best case (but also other strategies available; TBD)
- Do Partition based on the *median* (ensuring 2 equal halves)
- Does this affect f(n) (we should stay within O(n))

Median selection

- Put QS on hold
- Solve a more general problem – **selection problem** = given an unordered array, find the element which in the ordered array would occur in the i^{th} position (obviously, without ordering the array)
- Median selection = selection when $i=n/2$

Selection (generalization of *median* selection)

- Selects the i^{th} smallest element from an unordered array
- TBD on trees as well
- Hoare's algorithm
- Resembles the QuickSort algorithm, but with just one recursive call

i^{th} Selection - code

Selects *i^{th} smallest element* in A

(when i=n/2 => median selection)

```
Select(A,p,r,i)          //p=first, r=last, i=desired
  if p=r           //got the ith element in the right place
     then return A[p]
  q<-partition(A,p,r)      // q =index of the position
                           //where the partition stops
  k<-q-p+1             //k=length of the left partition
  if i<=k
     then return Select(A,p,q-1,i)
     else return Select(A,q+1,r,i-k)
```

Selection – eval - **Ω(n)**

- Cases are similar to QuickSort, yet just a single recursive call
- Worst

  $t(n)=n+(n-1)+(n-2)+\ldots=O(n^2)$ => **NOT** optimal
- Average

  $t(n)=n+n/2+\ldots=O(n)$
- Best

  Element found after a single partition pass (no recursive call) =>O(n)

Optimal Selection

- The same situation as for QuickSort: need to avoid worst case!
- Akl's alg = derived from parallel processing
- Splits the input data into a sub-arrays such that the selection is optimal

Selection – alg. description

Selection (A[1,n],i)

1. Split the array into sub-arrays of dim **a** each A_i, i=1,n/a.
2. Direct sort each A_i, and find its median, $\mathbf{m_i}$.
3. Generate the array of medians, and call the *Selection(m[1,n/a],n/a)* alg on the new array, to select the median of medians (i.e. M=m[n/a]).
4. Partition the input array into elements <= and >= M respectively. Assume there are k elements <=M.
5. **if** i<=k

   then *Selection (A[1,k],i)*

   else *Selection (A[k+1,n],i-k)*

Selection – alg. eval

- Determine *a* such that the alg is optimal
- $\Omega(n)$ => it should be $O(n)$
- Assume $t(n)$ the running time
- The steps:

1. a=ct=> $c_1 \cdot n$
2. O(1) for one seq, n/a seqs=>$c_2 \cdot n$
3. rec call on n/a els => $t(n/a)$
4. Partition =>$c_4 \cdot n$
5. We claim it is $t(3n/4)$ (justification follows)

Selection – alg. eval

We have: $t(n)=c \cdot n+t(n/a)+t(3n/4)$ (1)

We need: $t(n)<=k \cdot n$ (2)

Therefore:

$t(n) \quad = \quad c \cdot n+t(n/a)+t(3n/4)$

$\quad <= \quad \mathbf{c \cdot n+k \cdot n/a+k \cdot 3n/4<=k \cdot n}$ (3)

$=> \quad c \cdot n <=k \cdot (1/4-1/a) \cdot n$

$c>0$, $a>0 => ¼-1/a>0 =>a>4=>\mathbf{a_{min}=5}$

For a=5, we have that $\exists c$ s.t. $t(n)=c \cdot n$ =>**O(n) => OPTIMAL!**

Selection – alg. eval

- Why is step #5 t(3n/4) at most?
- M<= half of m_i's =>∃**n/2a** m_i's such that

$$\mathbf{m_i>=M \quad (1)}$$

- Each median m_i is **>= and <=** than exactly half of the nb. of elements in A_i, hence ∃**a/2** Ai's such that

$$\mathbf{m_i<=A_i \quad (2)}$$

(1)=>M is <=than n/2a medians m_i

(2)=>Each such median <= a/2 elements

Overall: M<= than at least **n/2a· a/2**=**n/4** elements

Selection – alg. eval

- With a similar reasoning, M>= than at least n/4 elements
- How are the rest?
  - Unknown!
- So?
  - The longest rec call is on 3n/4
- Cl: Alg is optimal for a>=5
- In practice, for parallel exe, a=8 (or another power of 2; depends on the nb. of processing units available)

Median selection

- Over
- May use it in QS
  - its optimal version has O(n)
  - by median partition, QS enters best case always
- Resume QS

QuickSort revised

```
QuickSort(A,p,r)
if p<r
   then    q<-partition(A,p,r)
           QuickSort(A,p,q)
           QuickSort(A,q+1,r)
```

- Worst case running time: $O(n^2)$ due to uneven partitioning
- Avoid worst case: use the “right” partitioning sequence (i.e. split input data into 2 equal subsets)

QuickSort revised – cont.

- Element to split the input data = median
  (i.e. element which in the ordered array would occur in the middle)
- Use a Median Selection **before** partitioning (we'll see shortly that's actually **instead** of partitioning)
- Selection – revised
  - Hoare's alg.
    - kind of QS with only 1 recursive call
    - inefficient $O(n^2)$ worst case running time; no improvement
  - Akl's alg (the one described before)
    - Optimal for a>=5 => O(n)
    - Multiplicative ct. very large (i.e. in the average case, Hoare's alg. is much better!)

QuickSort revised - transformation with selection

```
QuickSort(A,p,r)
if p<r
  then
  Select(A,p,r,|A|/2)
  q<-partition(A,p,r)//use the element returned by Select
  QuickSort(A,p,q)
  QuickSort(A,p,|A|/2)
  QuickSort(A,q+1,r)
  QuickSort(A,|A|/2+1,r)
```

Q: what is the effect of partition?
Is it required any more?
Note: partitioning and the blue QS calls get out

QuickSort transformed

```
QuickSort(A,p,r)

if p<r

   then
   Select(A,p,r,|A|/2)  //determines the median, and
                        //partitions based on the median
   QuickSort(A,p,|A|/2)
   QuickSort(A,|A|/2+1,r)
```

- How many rec. calls?
- Half done on leaves (i.e. empty data structures, thus call and return – takes time for doing nothing)
- What is the efficiency of rec. calls on small data structures?
- Avoid rec. calls on small data.

QuickSort enhanced

```
QuickSort(A,p,r)
if (r-p)<δ
  then   direct_sort(A,p,r)//which one?
  else
     Select(A,p,r,|A|/2)
     QuickSort(A,p,|A|/2)
     QuickSort(A,|A|/2+1,r)
```

Enhancements

p-r<δ saves time (secs, overhead of calls/restores from calls),

Select ensures the optimality (always falling into the best case) of the alg

QuickSort second revision

- In previous version `Select` call guarantees best case always
- QuickSort is O(nlgn) in the average case
- **It's enough to avoid the worst case**
- A **random** partition ensures this!
- Before partitioning, at each step pick a **random** element to make the partitioning based on that element (so swap the random chosen element with the element placed in the position of the pivot – first/middle/last)

QuickSort second revision– cont.

```
random_partition(A,p,r)
i<-random(p,r)              //choose a random element
A[i]<->A[p]                 //put it in the first position
return partition(A,p,r)     //here we have the
                            // regular one

QuickSort-Random(A,p,r)

if p<r

  then    q<-random_partition(A,p,r)
          QuickSort(A,p,q)
          QuickSort(A,q+1,r)
```

QuickSort second revision enhanced

QuickSort-Random(A,p,r)

```
if (r-p)<δ
   then     direct_sort(A,p,r)
   else     q<-random_partition(A,p,r)
            QuickSort(A,p,q)
            QuickSort(A,q+1,r)
```

Sorting - conclusions

- No direct method is optimal; all are $O(n^2)$, even if some behave well in best/average cases
- **Heapsort – is optimal**
- Heaps often used in **Priority Queues**
- QuickSort
  - classic version not optimal
  - Improved versions optimal:
    - Choose a **random** element to make the split
    - Use an **optimal selection** alg. (Akl's) to find the "split" point
    - Augment the alg with a direct method for small arrays, s.t. improve time (in secs, not t(n))