Fundamental Algorithms
Lecture #3

Agenda

- Master Theorem – to be remembered
- Features to evaluate – review
- Heap structure - review
- QuickSort
- i^{th} 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. **$q<1$; $a<b^c$ => $O(n^c)$**
2. **$q=1$; $a=b^c$ => $O(n^c * \log_b n)$**
3. **$q>1$; $a>b^c$ => $O(n^{\log_b a})$**

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
- push_heap
  - Increase the heap_size
  - Adds a new element at the bottom
  - 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)

Heap – as a data structure – cont.

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

Heap – comparison in building the heap

Approach	**Sol 1** (heapify)	**Sol2**(pop/push)
1 el approach	sinks the root O(h)	bubbles a leaf O(h)
all els(build heap) approach	bottom-up (starts with the last non-leaf 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 if 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): Master theorem: 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])
                 else return j
   end
```

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
         repeat   j<-j-1
         until    A[j]<=x
         repeat   i<-i+1
         until    A[i]>=x
         if i<j
                  then swap (A[i],A[j])
                  else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	9	3	12	5	7	2	9	5

i=0 *j=9*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]   i<-p-1    j<-r+1
while i<=j do
   begin
         repeat   j<-j-1
         until    A[j]<=x
         repeat   i<-i+1
         until    A[i]>=x
         if i<j
                  then swap (A[i],A[j])
                  else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	9	3	12	5	7	2	9	5

i=0 ***j=8***

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]   i<-p-1    j<-r+1
while i<=j do
    begin
          repeat   j<-j-1
          until    A[j]<=x
          repeat   i<-i+1
          until    A[i]>=x
          if i<j
                   then swap (A[i],A[j])
                   else return j
    end
```

x=9

	1	2	3	4	5	6	7	8
A	9	3	12	5	7	2	9	5

i=1 *j=8*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]   i<-p-1    j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                then swap (A[i],A[j])
                else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	12	5	7	2	9	9

i=1 *j=8*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                 then swap (A[i],A[j])
                 else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	12	5	7	2	9	9
	i=1						*j=7*	

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
         repeat   j<-j-1
         until    A[j]<=x
         repeat   i<-i+1
         until    A[i]>=x
         if i<j
                  then swap (A[i],A[j])
                  else return j
   end
```

x=9

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
A	5	3	12	5	7	2	9	9
			i=3				***j=7***	

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]   i<-p-1    j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                 then swap (A[i],A[j])
                 else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	9	5	7	2	12	9

i=3 *j=7*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                 then swap (A[i],A[j])
                 else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	9	5	7	2	12	9

i=3 *j=6*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                 then swap (A[i],A[j])
                 else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	9	5	7	2	12	9

j=6 *i=7*

Partition (Hoare original)

```
Partition(A,p,r)
x<-A[p]  i<-p-1   j<-r+1
while i<=j do
   begin
        repeat   j<-j-1
        until    A[j]<=x
        repeat   i<-i+1
        until    A[i]>=x
        if i<j
                 then swap (A[i],A[j])
                 else return j
   end
```

x=9

	1	2	3	4	5	6	7	8
A	5	3	9	5	7	2	12	9

j=6 ***i=7***

…returns 6

Partition (Hoare original)

```
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])
            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. Why? How can it 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 non-strict 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 – evaluation

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 – evaluation – contd.

- Not an optimal algorithm: $O(n^2) > \Omega(n \cdot \lg n)$
- $O(n\lg n)$ 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
    - Do Partition based on the *median* (ensuring 2 equal halves)
    - Does this affect f(n) (we should stay within O(n))
  - Randomization (TBD)

Median selection

Put QS on hold for now…

- **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 $i=n/2$

i^{th} Selection

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

i^{th} Selection – code

```
QuickSelect(A,p,r,i) //p=first, r=last, i=desired rank
  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, rank of the pivot
  if i=k
     then return A[q]
  else if i<k
     then return QuickSelect(A,p,q-1,i)
  else return QuickSelect(A,q+1,r,i-k)
```

QuickSelect – evaluation

- Problem lower bound: $\boldsymbol{\Omega(n)}$[1]
- 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) =>$O(n)$

[1] https://jeffe.cs.illinois.edu/teaching/497/02-selection.pdf

Optimal Selection

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

AklSelection

AklSelection (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 *AklSelection(m[1,n/a],n/a)* 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 *AklSelection(A[1,k],i)*

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

AklSelection – algorithm evaluation

- **Determine *a* such that the alg. is optimal**
- **Ω(n)** => it should be O(n)
- Assume t(n) the running time
- The steps:
  1. (split) a=constant => $\mathbf{c_1 \cdot n}$
  2. (sort) O(1) for one seq, n/a seqs=>$\mathbf{c_2 \cdot n}$
  3. (rec. call on n/a elements) => **t(n/a)**
  4. (partition) =>$\mathbf{c_4 \cdot n}$
  5. (rec. call on one partition) => at most **t(3n/4)** (justification follows in 2 slides)

AklSelection – alg. eval. – contd.

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

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

Therefore:

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

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

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

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

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

OPTIMAL!

AklSelection – alg. eval. – contd.

- 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

AklSelection – alg. eval. – contd.

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

Median selection

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

QuickSort revised (rv1)

```
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 (rv1) – 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 (rv1) - transformation with selection

```
QuickSort(A,p,r)
if p<r
  then
  AklSelect(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 rv1

```
QuickSort(A,p,r)
if p<r
   then
   AklSelect(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 rv1 enhanced

```
QuickSort(A,p,r)
if (r-p)<δ
  then   direct_sort(A,p,r)//which one?
  else
     AklSelect(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 revised (rv2)

- In rv1, **`AklSelect`** 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 rv2 – 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 rv2 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)
```

Merge Sort

- Divide et impera
- Attempts (and succeeds) to always enter the best case
- Approach (see blackboard)
  - Divides the input into 2 equal partitions (chooses the middle)
  - Apply 2 recursions
  - Merge the resulting ordered halves
- Master Theorem: a= 2; b=2; c=1 => O(nlgn). Is it optimal? Why/why not?
- Where does it apply?

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))

Required Bibliography

- From the Bible – Chapter 7 (QuickSort), Sections 9.2 and 9.3 (Selection problem algorithms)