Lecture #3 Sorting. QuickSort

Fundamental Algorithms

Rodica Potolea, Camelia Lemnaru și Ciprian Oprișa

Technical University of Cluj-Napoca
Computer Science

October 2024

Agenda

1. Master Theorem - review
2. Algorithm features to evaluate - review
3. QuickSort
4. i^{th} selection

Agenda

Master Theorem - review

$$T(n) = \begin{cases} T_0, & \text{if } n < n_0 \\ a * T(\frac{n}{b}) + n^c, & \text{otherwise} \end{cases} \quad (1)$$

where ...

- a: # of *recursive calls*
- b: *division factor* = ratio between original size and recursive size
- c: *degree of polynomial* of the execution time of the sequence *excepting the recursive calls*: $f(n) = n^c$

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

Agenda

Algorithm features to evaluate - review

- Correctness
  - *Partial* vs *total*
- Efficiency vs optimality
  - Cases depend on
    - the *problem* being solved
    - the *algorithm* solving the problem
    - the *implementation* of the algorithm
- Stability
  - *Stable* vs *unstable* algorithm
- Determinism
  - *Deterministic* vs *nondeterministic* behavior

Agenda

1. Master Theorem - review
2. Algorithm features to evaluate - review
3. QuickSort
4. i^{th} selection

QuickSort

- In a nutshell...
  - base (vanilla) algorithm not optimal
  - better than *Heapsort* in practice
  - can be made optimal (sort in at most $O(nlgn)$ time, with constant additional space)

QuickSort

```
QuickSort(A, p, r)
if p < r // if non empty array
    q = Partition(A, p, r)
    // q is an index, boundary between the two partitions
    QuickSort(A, p, q)
    QuickSort(A, q + 1, r)
```

- Why are the two partitions like that?
- Where is the pivot?

Partition (Hoare)

HOARE-PARTITION(A, p, r)

```
x = A[p]
i = p - 1
j = r + 1
while true
    repeat
        j = j - 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

Partition (Hoare)

```
HOARE-PARTITION(A, p, r)
x = A[p]
i = p − 1
j = r + 1
while true
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

x=9

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

i=0 *j=9*

- What values will i and j have after the first **while** iteration ?
- How will A look after the first **while** iteration ?

Partition (Hoare)

HOARE-PARTITION(A, p, r)

```
x = A[p]
i = p − 1
j = r + 1
while true
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

x=9

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

- What values will i and j have after the first **while** iteration ?
- How will A look after the first **while** iteration ?

Partition (Hoare)

HOARE-PARTITION(A, p, r)

```
x = A[p]
i = p − 1
j = r + 1
while true
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

x=9

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

i=1 *j=8*

- How will the array look after the algorithm returns?
- What will the algorithm return?

Partition (Hoare)

HOARE-PARTITION(A, p, r)

```
x = A[p]
i = p − 1
j = r + 1
while true
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

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*

- How will the array look after the algorithm returns?
- What will the algorithm return?

Partition (Hoare)

HOARE-PARTITION(A, p, r)

```
x = A[p]
i = p − 1
j = r + 1
while true
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

!! *Homework* !!

- i and j never go out of the array boundaries. Why?
- **repeat-until** loops stop on equal elements and swaps them. Why?
- $A[p]$ as pivot has an undesired worst case (leads QuickSort to $O(n^2)$). Which is it? Why is it undesired?
- $A[p]$ pivot is essential for correctness. Why? (e.g. $A[r]$ as pivot causes execution error. Why?)
- $T(n) = ?$

Partition (Hoare Update)

```
HOARE-PARTITION-UPDATE(A, p, r)
x = A[(p + r)/2]
i = p
j = r
repeat
    while A[i] < x
        i = i + 1
    while A[j] > x
        j = j - 1
    if i ≤ j
        exchange A[i] with A[j]
        i = i + 1
        j = j - 1
until i < j
return j
```

- other pivot choices possible, but small adjustments needed (*Hw*)

Partition (Hoare Update)

HOARE-PARTITION-UPDATE(A, p, r)

```
x = A[(p + r)/2]
i = p
j = r
repeat
    while A[i] < x
        i = i + 1
    while A[j] > x
        j = j - 1
    if i ≤ j
        exchange A[i] with A[j]
        i = i + 1
        j = j - 1
until i < j
return j
```

- other pivot choices possible, but small adjustments needed (*Hw*)
- Trace the execution on the array $A = \{9, 3, 12, 5, 7, 2, 9, 5\}$ (*Hw*)

Partition (Hoare Update)

HOARE-PARTITION-UPDATE(A, p, r)

```
x = A[(p + r)/2]
i = p
j = r
repeat
        while A[i] < x
                i = i + 1
        while A[j] > x
                j = j - 1
        if i ≤ j
                exchange A[i] with A[j]
                i = i + 1
                j = j - 1
until i < j
return j
```

- other pivot choices possible, but small adjustments needed (*Hw*)
- Trace the execution on the array $A = \{9, 3, 12, 5, 7, 2, 9, 5\}$ (*Hw*)
- **while** loop stops on equal elements and swaps them. Can they be left in the original partition? (i.e. use non-strict inequalities)

Partition (Hoare Update)

HOARE-PARTITION-UPDATE(A, p, r)

```
x = A[(p + r)/2]
i = p
j = r
repeat
    while A[i] < x
        i = i + 1
    while A[j] > x
        j = j - 1
    if i ≤ j
        exchange A[i] with A[j]
        i = i + 1
        j = j - 1
until i < j
return j
```

- other pivot choices possible, but small adjustments needed (*Hw*)
- Trace the execution on the array $A = \{9, 3, 12, 5, 7, 2, 9, 5\}$ (*Hw*)
- **while** loop stops on equal elements and swaps them. Can they be left in the original partition? (i.e. use non-strict inequalities)
- if $i = j$, elements are swapped. Redundant?

Partition (Hoare Update)

HOARE-PARTITION-UPDATE(A, p, r)

```
x = A[(p + r)/2]
i = p
j = r
repeat
    while A[i] < x
        i = i + 1
    while A[j] > x
        j = j - 1
    if i ≤ j
        exchange A[i] with A[j]
        i = i + 1
        j = j - 1
until i < j
return j
```

- other pivot choices possible, but small adjustments needed (*Hw*)
- Trace the execution on the array $A = \{9, 3, 12, 5, 7, 2, 9, 5\}$ (*Hw*)
- **while** loop stops on equal elements and swaps them. Can they be left in the original partition? (i.e. use non-strict inequalities)
- if $i = j$, elements are swapped. Redundant?
- $T(n) = ?$

QuickSort evaluation

```
QUICKSORT(A, p, r)
  if p < r // if non empty array
      q = PARTITION(A, p, r)
      // q is an index, boundary between the two partitions
      QUICKSORT(A, p, q)
      QUICKSORT(A, q + 1, r)
```

- $a = ?$
- $b = ?$
- $c = ?$

QuickSort evaluation

```
QUICKSORT(A, p, r)
if p < r // if non empty array
    q = PARTITION(A, p, r)
    // q is an index, boundary between the two partitions
    QUICKSORT(A, p, q)
    QUICKSORT(A, q + 1, r)
```

- $a = 2$
- $b =$ ... depends on the case. On what, specifically?
- $c = 1$

QuickSort evaluation

$\textsc{QuickSort}(A, p, r)$

if $p < r$ // if non empty array
    $q = \textsc{Partition}(A, p, r)$
    // q is an index, boundary between the two partitions
    $\textsc{QuickSort}(A, p, q)$
    $\textsc{QuickSort}(A, q + 1, r)$

- $a = 2$
- $b = \ldots$ depends on the pivot choice, so on the *implementation*!
- $c = 1$

QuickSort evaluation

b depends on the pivot choice, so on the *implementation*!

- **best** case: $b = 2$ (2 equal partitions); $T(n) = O(nlgn)$
- **average** case: $b = 2$ – can be shown, on average, the partitions are balanced enough; $T(n) = O(nlgn)$
- **worst** case: a partition with 1 element, the other with $n - 1$ elements; $T(n) = O(n) + T(n-1)$ (why?)
  So ... $T(n) = O(n^2)$
- Additional memory?

QuickSort evaluation

- NOT optimal: $T(n) = O(n^2) > \Omega(nlgn)$
- BUT worst case occurs seldom
  - How seldom?
  - Property of the data to enter worst case?
    - What factor(s) impact the case?
    - How does it depend on the implementation?
- Can we ensure we NEVER enter the worst case?
  - Always enter best case (ensure balanced partitions, in $O(n)$) ... coming next
  - Randomization ... TBD

Agenda

1. Master Theorem - review
2. Algorithm features to evaluate - review
3. QuickSort
4. i^{th} selection

i^{th} selection

- putting QS on hold for now, to discuss about:
  - the **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 = particular instance of the problem, when $i = n/2$
  - Algorithms:
    - QuickSelect (Hoare); based on QuickSort (only 1 recursive call)
    - AklSelect - strategy, parallel processing; optimal!

QuickSelect (Hoare)

$\textsc{QuickSelect}(A, p, r, i)$

// p - first, r - last, i desired rank
if $p = r$ // we are on the correct array position
    return $A[p]$
$q = \textsc{Hoare-Partition}(A, p, r)$ // q - index where partition ends
$k = q - p + 1$ // k - length of the $\leq$ partition
if $i \leq k$
    return $\textsc{QuickSelect}(A, p, q, i)$
else return $\textsc{QuickSelect}(A, q + 1, r, i - k)$

- Why only 1 recursive call?

QuickSelect (Hoare)

```
QUICKSELECT(A, p, r, i)
1  // p - first, r - last, i desired rank
2  if p = r // we are on the correct array position
3       return A[p]
4  q = HOARE-PARTITION(A, p, r) // q - index where partition ends
5  k = q − p + 1 // k - length of the ≤ partition
6  if i ≤ k
7       return QUICKSELECT(A, p, q, i)
8  else return QUICKSELECT(A, q + 1, r, i − k)
```

- Why only 1 recursive call?
- Why $i - k$ on recursive call in line 8?

QuickSelect (Hoare)

$\textsc{QuickSelect}(A, p, r, i)$

1 // p - first, r - last, i desired rank
2 **if** $p = r$ // we are on the correct array position
3   **return** $A[p]$
4 $q = \textsc{Hoare-Partition}(A, p, r)$ // q - index where partition ends
5 $k = q - p + 1$ // k - length of the $\leq$ partition
6 **if** $i \leq k$
7   **return** $\textsc{QuickSelect}(A, p, q, i)$
8 **else return** $\textsc{QuickSelect}(A, q + 1, r, i - k)$

- Why only 1 recursive call?
- Why $i - k$ on recursive call in line 8?
- Trace execution for $\textsc{QuickSelect}(A, 1, 8, 3)$, for $A = \{4, 8, 1, 9, 3, 4, 2, 6\}$

QuickSelect (Hoare)

```
QuickSelect(A, p, r, i)
// p - first, r - last, i desired rank
if p = r // we are on the correct array position
        return A[p]
q = Hoare-Partition(A, p, r) // q - index where partition ends
k = q − p + 1 // k - length of the ≤ partition
if i ≤ k
        return QuickSelect(A, p, q, i)
else return QuickSelect(A, q + 1, r, i − k)
```

- What changes need to be done to use Lomuto-Partition(A, p, r) instead? (Hw)

QuickSelect (Hoare) evaluation

$\textsc{QuickSelect}(A, p, r, i)$

```
// p - first, r - last, i desired rank
if p = r // we are on the correct array position
        return A[p]
q = HOARE-PARTITION(A, p, r) // q - index where partition ends
k = q - p + 1 // k - length of the partition
if i ≤ k
        return QUICKSELECT(A, p, q, i)
else return QUICKSELECT(A, q + 1, r, i - k)
```

- $a = ?$
- $b = ?$
- $c = ?$

QuickSelect (Hoare) evaluation

QuickSelect(A, p, r, i)

// p - first, r - last, i desired rank
if $p = r$ // we are on the correct array position
    return $A[p]$
q = Hoare-Partition(A, p, r) // q - index where partition ends
$k = q - p + 1$ // k - length of the partition
if $i \leq k$
    return QuickSelect(A, p, q, i)
else return QuickSelect($A, q + 1, r, i - k$)

- $a = 1$
- $b = ...$ depends on the case. On what, specifically?
- $c = 1$

QuickSelect (Hoare) evaluation

$\textsc{QuickSelect}(A, p, r, i)$

```
// p - first, r - last, i desired rank
if p = r // we are on the correct array position
      return A[p]
q = HOARE-PARTITION(A, p, r) // q - index where partition ends
k = q − p + 1 // k - length of the partition
if i ≤ k
      return QUICKSELECT(A, p, q, i)
else return QUICKSELECT(A, q + 1, r, i − k)
```

- $a = 1$
- $b = \ldots$ depends on the pivot choice, so on the *implementation*!
- $c = 1$

QuickSelect (Hoare) evaluation

- Problem lower bound: $\Omega(n)$ [1]
- **best** case: element found after a single partition pass: $T(n) = O(n)$ - how?
- **average** case: $T(n) = n + n/2 + n/4 + ... = O(n)$
- **worst** case: $T(n) = O(n) + T(n-1)$ (why?)
  So ... $T(n) = O(n^2)$ - NOT optimal
- Additional memory?

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

Akl's Algorithm

- derived from parallel processing, strategy rather than algorithm
- idea - split data into sub-arrays, to make selection optimal

Akl's Algorithm

$\textsc{AklSelect}(A[1..n], i)$

Split the array into i sub-arrays of size a, each: $A_i, i = 1 \rightarrow n/a$
Direct sort each A_i, and find its median, m_i.
Generate the array of medians, and call the
$\textsc{AklSelect}(m[1, n/a], n/2a)$ on the new array,
to select the median of medians (i.e. $M = m[n/a]$).
Partition the input array into elements $\leq M$ and $\geq M$, respectively.
Assume there are k elements $\leq M$.
if $i = k$
    return M
if $i < k$
    $\textsc{AklSelect}(A[1...k-1], i)$
else $\textsc{AklSelect}(A[k+1...n], i-k)$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
    - 1. (split), a - cst.:

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.:

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems):

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition):

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition):

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: T(n) $=$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: T(n) $=$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$
- need $T(n) \leq k * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: T(n) $=$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$
- need $T(n) \leq k * n$
- need $c * n + T(n/a) + T(3n/4) \leq k * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: T(n) $=$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$
- need $c * n + k * n/a + k * 3n/4 \leq k * n$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: $\mathrm{T(n)} =$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$
- need $c * n + k * n/a + k * 3n/4 \leq k * n$
- considering $c > 0, a > 0$; solve $- > a_{min} = 5$

AklSelect evaluation

- Problem lower bound: $\Omega(n)$
- Determine a such that the algorithm is optimal
- According to the algorithm steps: T(n) $=$
  - 1. (split), a - cst.: $c_1 * n$
  - 2. (sort), a - cst.: $O(1)$ for 1 seq., n/a seqs, so: $c_2 * n$
  - 3. (rec. call on n/a elems): $T(n/a)$
  - 4. (partition): $c_4 * n$
  - 7-9. (rec. call on one partition): at most $T(3n/4)$ (justification in 2 slides)
- $T(n) = c * n + T(n/a) + T(3n/4)$
- need $c * n + k * n/a + k * 3n/4 \leq k * n$
- considering $c > 0, a > 0$; solve $- > a_{min} = 5$
- So, for $a \geq 5, \exists c$ s.t. $T(n) = O(n)$, so it is OPTIMAL

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2\ A_i s$ s.t. $m_i \leq A_i$ (2)

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2\ A_i s$ s.t. $m_i \leq A_i$ (2)
- (1) $\implies M \leq n/2a$ medians m_i

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s \Rightarrow \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2\ A_i s$ s.t. $m_i \leq A_i$ (2)
- (1) $\implies$ $M \leq n/2a$ medians m_i
- (2) $\implies$ each such median $m_i \leq a/2$ elements A_i

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2\ A_i s$ s.t. $m_i \leq A_i$ (2)
- (1) $\implies M \leq n/2a$ medians m_i
- (2) $\implies$ each such median $m_i \leq a/2$ elements A_i
- Overall: $M \leq n/2a * a/2 = n/4$ elements

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a\ m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2\ A_i s$ s.t. $m_i \leq A_i$ (2)
- (1) $\implies M \leq n/2a$ medians m_i
- (2) $\implies$ each such median $m_i \leq a/2$ elements A_i
- Overall: $M \leq n/2a * a/2 = n/4$ elements
- Using a similar reasoning: $M \geq n/4$ elements

AklSelect evaluation

- Why is the effort in steps 7-9. at most $T(3n/4)$?
- $M \leq$ half of $m_i s => \exists n/2a$ $m_i s$ s.t. $m_i \geq M$ (1)
- each median m_i is $\leq$ and $\geq$ exactly half of the number of elements in A_i, hence $\exists a/2$ $A_i s$ s.t. $m_i \leq A_i$ (2)
- (1) $\implies$ $M \leq n/2a$ medians m_i
- (2) $\implies$ each such median $m_i \leq a/2$ elements A_i
- Overall: $M \leq n/2a * a/2 = n/4$ elements
- Using a similar reasoning: $M \geq n/4$ elements
- The other at most $3n/4$ are unknown $\implies$ longest recursive call is on $3n/4$ elements

QuickSort improvements (1)

QuickSortV2(A, p, r)

```
if p < r // if non empty array
    AklSelect(A, p, r, |A|/2)
    // determines the median, and partitions based on the median
    QuickSortV2(A, p, |A|/2)
    QuickSortV2(A, |A|/2 + 1, r)
```

- Avoid uneven partitioning by always splitting by the median

QuickSort improvements (1)

$\textsc{QuickSortV2}(A, p, r)$

```
if p < r // if non empty array
    AklSelect(A, p, r, |A|/2)
    // determines the median, and partitions based on the median
    QuickSortV2(A, p, |A|/2)
    QuickSortV2(A, |A|/2 + 1, r)
```

- Avoid uneven partitioning by always splitting by the median
    - QuickSelect?
    -

QuickSort improvements (1)

$\textsc{QuickSortV2}(A, p, r)$

if $p < r$ // if non empty array
    $\textsc{AklSelect}(A, p, r, |A|/2)$
    // determines the median, and partitions based on the median
    $\textsc{QuickSortV2}(A, p, |A|/2)$
    $\textsc{QuickSortV2}(A, |A|/2 + 1, r)$

- Avoid uneven partitioning by always splitting by the median
    - QuickSelect still has $O(n^2)$ worst case running time

QuickSort improvements (1)

QuickSortV2(A, p, r)

if $p < r$ // if non empty array
    AklSelect($A, p, r, |A|/2$)
    // determines the median, and partitions based on the median
    QuickSortV2($A, p, |A|/2$)
    QuickSortV2($A, |A|/2 + 1, r$)

- Avoid uneven partitioning by always splitting by the median
    - QuickSelect still has $O(n^2)$ worst case running time
    - AklSelect?

QuickSort improvements (1)

$\textsc{QuickSortV2}(A, p, r)$

```
if p < r // if non empty array
        AklSelect(A, p, r, |A|/2)
        // determines the median, and partitions based on the median
        QuickSortV2(A, p, |A|/2)
        QuickSortV2(A, |A|/2 + 1, r)
```

- Avoid uneven partitioning by always splitting by the median
  - QuickSelect still has $O(n^2)$ worst case running time
  - AklSelect optimal for $a \geq 5$ but very large multiplicative constant!

QuickSort improvements (1)

$\textsc{QuickSortV2}(A, p, r)$

if $p < r$ // if non empty array
    $\textsc{AklSelect}(A, p, r, |A|/2)$
    // determines the median, and partitions based on the median
    $\textsc{QuickSortV2}(A, p, |A|/2)$
    $\textsc{QuickSortV2}(A, |A|/2 + 1, r)$

- Avoid uneven partitioning by always splitting by the median
  - QuickSelect still has $O(n^2)$ worst case running time
  - AklSelect optimal for $a \geq 5$ but very large multiplicative constant!
  - QuickSelect is much better on average than AklSelect!

QuickSort improvements (1) + hybridization

```
QUICKSORTV21(A, p, r)
if r − p < ϵ // if non empty array
    DIRECTSORT(A, p, r) // which one?
else AKLSELECT(A, p, r, |A|/2)
    // determines the median, and partitions based on the median
    QUICKSORTV21(A, p, |A|/2)
    QUICKSORTV21(A, |A|/2 + 1, r)
```

- hybridization saves time from the overhead of calls/restores from calls (call stack operations)

QuickSort improvements (2) + hybridization

$\textsc{QuickSortV3}(A, p, r)$

```
if r − p < ϵ // if non empty array
    DirectSort(A, p, r)
else q = RandomPartition(A, p, r)
    QuickSortV3(A, p, q)
    QuickSortV3(A, q + 1, r)
```

- In V2, AklSelect guarantees best partitioning always, but with large constant increase

QuickSort improvements (2) + hybridization

$\textsc{QuickSortV3}(A, p, r)$

if $r - p < \epsilon$ // if non empty array
    $\textsc{DirectSort}(A, p, r)$
else $q = \textsc{RandomPartition}(A, p, r)$
    $\textsc{QuickSortV3}(A, p, q)$
    $\textsc{QuickSortV3}(A, q + 1, r)$

- In V2, AklSelect guarantees best partitioning always, but with large constant increase
- QuickSort has a very low constant in the average case

QuickSort improvements (2) + hybridization

$\textsc{QuickSortV3}(A, p, r)$

if $r - p < \epsilon$ // if non empty array
    $\textsc{DirectSort}(A, p, r)$
else $q = \textsc{RandomPartition}(A, p, r)$
    $\textsc{QuickSortV3}(A, p, q)$
    $\textsc{QuickSortV3}(A, q + 1, r)$

- In V2, AklSelect guarantees best partitioning always, but with large constant increase
- QuickSort has a very low constant in the average case
- So, avoid the worst case

QuickSort improvements (2) + hybridization

$\textsc{QuickSortV3}(A, p, r)$

```
if r − p < ϵ // if non empty array
        DirectSort(A, p, r)
else q = RandomPartition(A, p, r)
        QuickSortV3(A, p, q)
        QuickSortV3(A, q + 1, r)
```

- In V2, AklSelect guarantees best partitioning always, but with large constant increase
- QuickSort has a very low constant in the average case
- So, avoid the worst case
- A random partition ensures this!

QuickSort improvements (2) + hybridization

QUICKSORTV3(A, p, r)

if $r - p < \epsilon$ // if non empty array
        DIRECTSORT(A, p, r)
else $q =$ RANDOMPARTITION(A, p, r)
        QUICKSORTV3(A, p, q)
        QUICKSORTV3$(A, q + 1, r)$

RANDOMPARTITION(A, p, r)

$i =$ RANDOM(p, r)
exchange $A[p]$ with $A[i]$
return HOARE-PARTITION(A, p, r)

MergeSort

$\textsc{MergeSort}(A, p, r)$

if $p \geq r$ // zero or one element?
    return
$q = \lfloor (p + r)/2 \rfloor$
$\textsc{MergeSort}(A, p, q)$
$\textsc{MergeSort}(A, q + 1, r)$
$\textsc{Merge}(A, p, q, r)$

- also uses *divide et impera*
- partitions always balanced
- $T(n) = 2T(n/2) + O(n)$
- NOT optimal. Why?
- When do we use it?

Sorting - conclusions

- No direct method is optimal; all are $O(n^2)$, even if some behave well in best case
- HeapSort is **optimal**
- Heaps used to implement priority queues
- QuickSort
  - classic version not optimal
  - improved versions are optimal
    - Choose a **random** pivot to make the split
    - Use an **optimal selection** alg. (Akl) to find the "split" point
    - Augment the alg. with a direct method for small arrays, to improve time (in secs, not $T(n)$)

Required Bibliography

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