7. Sorting I

Simple Sorting

7.1 Simple Sorting

Selection Sort, Insertion Sort, Bubblesort [Ottman/Widmayer, Kap. 2.1, Cormen et al, Kap. 2.1, 2.2, Exercise 2.2-2, Problem 2-2

Problem

Input: An array $A = (A[1], ..., A[n])$ with length n.

Output: a permutation A' of A, that is sorted: $A'[i] \leq A'[j]$ for all $1 \leq i \leq j \leq n$.

Algorithm: IsSorted(A)

Input: Array $A = (A[1], ..., A[n])$ with length n.
Output: Boolean decision "sorted" or "not sorted"

```
for i ← 1 to n − 1 do
    if A[i] > A[i + 1] then
        return "not sorted";
return "sorted";
```

Observation

IsSorted(A):"not sorted", if $A[i] > A[i+1]$ for any i.

$\Rightarrow$ idea:

for $j \leftarrow 1$ **to** $n-1$ **do**
  if $A[j] > A[j+1]$ **then**
    swap($A[j], A[j+1]$);

Give it a try

5	6	2	8	4	1	$(j=1)$
5	6	2	8	4	1	$(j=2)$
5	2	6	8	4	1	$(j=3)$
5	2	6	8	4	1	$(j=4)$
5	2	6	4	8	1	$(j=5)$
5	2	6	4	1	8	

- Not sorted! ☹.
- But the greatest element moves to the right
  $\Rightarrow$ new idea! ☺

Try it out

5	6	2	8	4	1	$(j=1, i=1)$
5	6	2	8	4	1	$(j=2)$
5	2	6	8	4	1	$(j=3)$
5	2	6	8	4	1	$(j=4)$
5	2	6	4	8	1	$(j=5)$
5	2	6	4	1	8	$(j=1, i=2)$
2	5	6	4	1	8	$(j=2)$
2	5	6	4	1	8	$(j=3)$
2	5	4	6	1	8	$(j=4)$
2	5	4	1	6	8	$(j=1, i=3)$
2	5	4	1	6	8	$(j=2)$
2	4	5	1	6	8	$(j=3)$
2	4	1	5	6	8	$(j=1, i=4)$
2	4	1	5	6	8	$(j=2)$
2	1	4	5	6	8	$(i=1, j=5)$
1	2	4	5	6	8	

- Apply the procedure iteratively.
- For $A[1,\dots,n]$, then $A[1,\dots,n-1]$, then $A[1,\dots,n-2]$, etc.

Algorithm: Bubblesort

Input: Array $A = (A[1],\dots,A[n])$, $n \geq 0$.
Output: Sorted Array A

for $i \leftarrow 1$ **to** $n-1$ **do**
  for $j \leftarrow 1$ **to** $n-i$ **do**
    if $A[j] > A[j+1]$ **then**
      swap($A[j], A[j+1]$);

Analysis

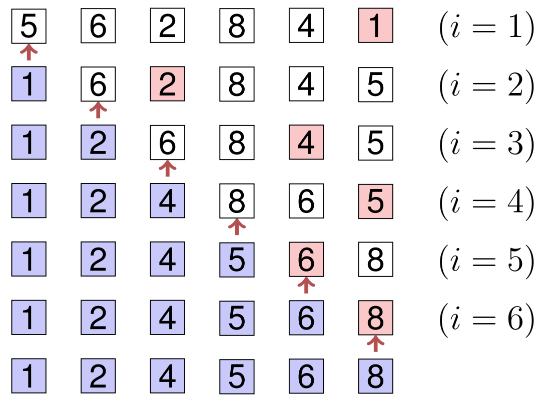
Number key comparisons $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$.

Number swaps in the worst case: $\Theta(n^2)$

? What is the worst case?

! If A is sorted in decreasing order.

Selection Sort



- Selection of the smallest element by search in the unsorted part $A[i..n]$ of the array.
- Swap the smallest element with the first element of the unsorted part.
- Unsorted part decreases in size by one element ($i \to i+1$). Repeat until all is sorted. ($i = n$)

Algorithm: Selection Sort

Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sorted Array A

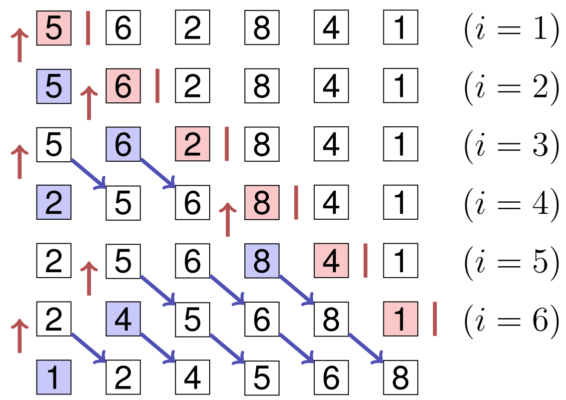
for $i \leftarrow 1$ **to** $n-1$ **do**
  $p \leftarrow i$
  for $j \leftarrow i+1$ **to** n **do**
    if $A[j] < A[p]$ **then**
      $p \leftarrow j$;
  swap($A[i], A[p]$)

Analysis

Number comparisons in worst case: $\Theta(n^2)$.

Number swaps in the worst case: $n - 1 = \Theta(n)$

Insertion Sort



- Iterative procedure: $i = 1...n$
- Determine insertion position for element i.
- Insert element i array block movement potentially required

Insertion Sort

? What is the disadvantage of this algorithm compared to sorting by selection?

! Many element movements in the worst case.

? What is the advantage of this algorithm compared to selection sort?

! The search domain (insertion interval) is already sorted. Consequently: binary search possible.

Algorithm: Insertion Sort

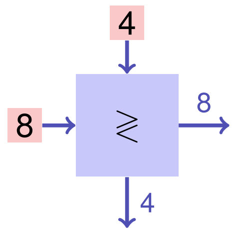
Input: Array $A = (A[1], \dots, A[n])$, $n \geq 0$.
Output: Sorted Array A
for $i \leftarrow 2$ **to** n **do**
    $x \leftarrow A[i]$
    $p \leftarrow$ BinarySearch($A[1...i-1], x$); // Smallest $p \in [1, i]$ with $A[p] \geq x$
    for $j \leftarrow i - 1$ **downto** p **do**
        $A[j+1] \leftarrow A[j]$
    $A[p] \leftarrow x$

Analysis

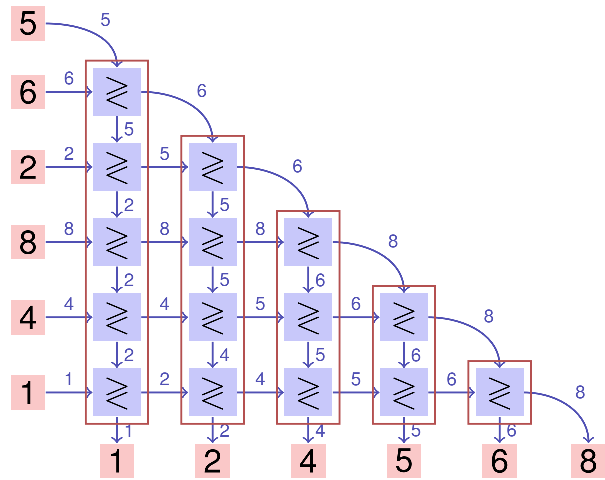
Number comparisons in the worst case:
$\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \mathcal{O}(n \log n)$.
Number swaps in the worst case $\sum_{k=2}^{n} (k-1) \in \Theta(n^2)$

Different point of view

Sorting node:

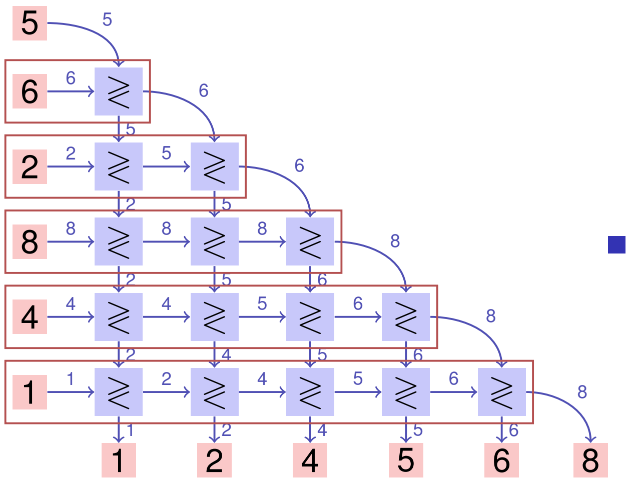


Different point of view



- Like selection sort
  [and like Bubblesort]

Different point of view



- Like insertion sort

Conclusion

In a certain sense, Selection Sort, Bubble Sort and Insertion Sort provide the same kind of sort strategy. Will be made more precise. [6]

[6] In the part about parallel sorting networks. For the sequential code of course the observations as described above still hold.

Shellsort (Donald Shell 1959)

Insertion sort on subsequences of the form $(A_{k \cdot i})$ $(i \in \mathbb{N})$ with decreasing distances k. Last considered distance must be $k = 1$.

Worst-case performance critically depends on the chosen subsequences

- Original concept with sequence $1, 2, 4, 8, ..., 2^k$. Running time: $\mathcal{O}(n^2)$
- Sequence $1, 3, 7, 15, ..., 2^{k-1}$ (Hibbard 1963). $\mathcal{O}(n^{3/2})$
- Sequence $1, 2, 3, 4, 6, 8, ..., 2^p 3^q$ (Pratt 1971). $\mathcal{O}(n \log^2 n)$

Shellsort

9	8	7	6	5	4	3	2	1	0	
1	8	7	6	5	4	3	2	9	0	insertion sort, $k = 4$
1	0	7	6	5	4	3	2	9	8	
1	0	3	6	5	4	7	2	9	8	
1	0	3	2	5	4	7	6	9	8	
1	0	3	2	5	4	7	6	9	8	insertion sort, $k = 2$
1	0	3	2	5	4	7	6	9	8	
0	1	2	3	4	5	6	7	8	9	insertion sort, $k = 1$

8. Sorting II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

Heapsort

Inspiration from selectsort: fast insertion

Inspiration from insertion sort: fast determination of position

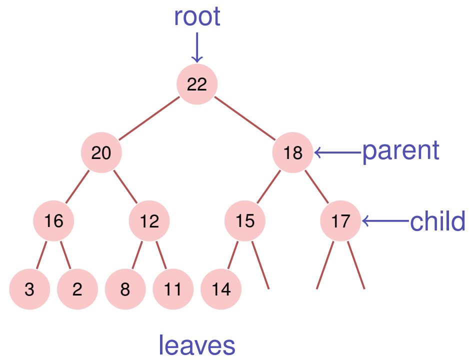
? Can we have the best of both worlds?

! Yes, but it requires some more thinking...

[Max-]Heap[7]

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. *Heap-Condition:* Max-(Min-)Heap: key of a child smaller (greater) that that of the parent node

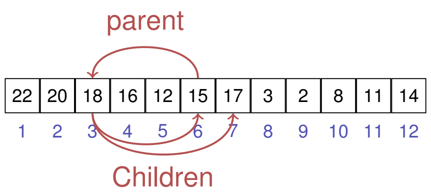


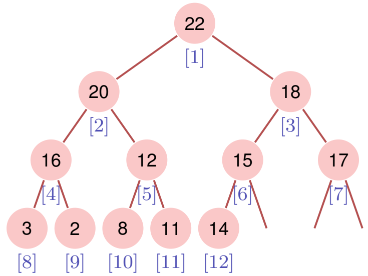
[7] Heap(data structure), not: as in "heap and stack" (memory allocation)

Heap as Array

Tree $\rightarrow$ Array:

- children$(i) = \{2i, 2i+1\}$
- parent$(i) = \lfloor i/2 \rfloor$



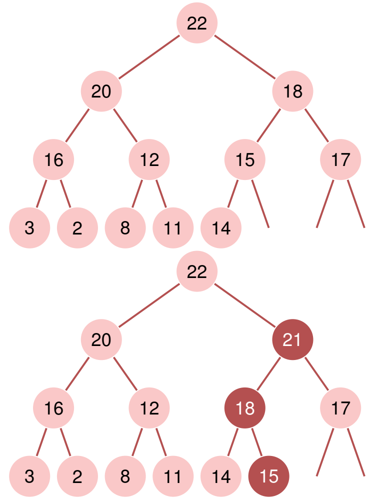


Depends on the starting index[8]

[8] For array that start at 0: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively
- Worst case number of operations: $\mathcal{O}(\log n)$



Algorithm Sift-Up(A, m)

Input: Array A with at least $m+1$ and Max-Heap-Structure on $A[0, \ldots, m-1]$
Output: Array A with Max-Heap-Structure on $A[0, \ldots, m]$.

$v \leftarrow A[m]$ // value
$c \leftarrow m$ // current position
$p \leftarrow \lfloor (c-1)/2 \rfloor$ // parent node
while $c > 0$ and $v > A[p]$ **do**
  $A[c] \leftarrow A[p]$ // Value parent node $\rightarrow$ current node
  $c \leftarrow p$ // parent node $\rightarrow$ current node
  $p \leftarrow \lfloor (c-1)/2 \rfloor$
$A[c] \leftarrow v$ // value $\rightarrow$ current node

Height of a Heap

A complete binary tree with height[9] h provides

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

nodes. Thus for a heap with height h:

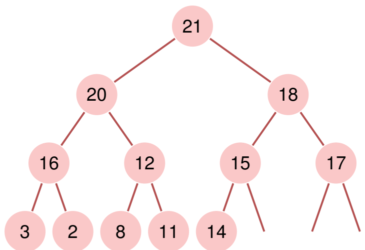
$$2^{h-1} - 1 < n \leq 2^h - 1$$
$$\Leftrightarrow \quad 2^{h-1} < n + 1 \leq 2^h$$

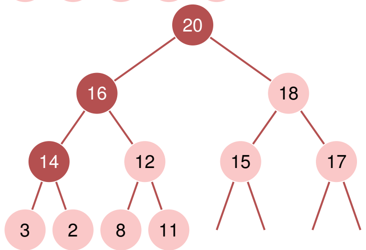
Particularly $h(n) = \lceil \log_2(n+1) \rceil$ and $h(n) \in \Theta(\log n)$.

[9] here: number of edges from the root to a leaf

Remove the maximum

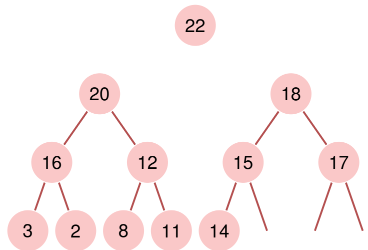
- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$





Why this is correct: Recursive heap structure

A heap consists of two heaps:



Algorithm SiftDown(A, i, m)

Input: Array A with heap structure for the children of i. Last element m.
Output: Array A with heap structure for i with last element m.

while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j left child
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j + 1$; // j right child with greater key
  if $A[i] < A[j]$ **then**
    swap($A[i], A[j]$)
    $i \leftarrow j$; // keep sinking down
  else
    $i \leftarrow m$; // sift down finished

Sort heap

$A[1, ..., n]$ is a Heap.
While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n-1$);
- $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7
siftDown	$\Rightarrow$	5	4	2	1	6	7
swap	$\Rightarrow$	1	4	2	5	6	7
siftDown	$\Rightarrow$	4	1	2	5	6	7
swap	$\Rightarrow$	2	1	4	5	6	7
siftDown	$\Rightarrow$	2	1	4	5	6	7
swap	$\Rightarrow$	1	2	4	5	6	7

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

Algorithm HeapSort(A, n)

Input: Array A with length n.
Output: A sorted.

// Build the heap.
for $i \leftarrow n/2$ **downto** 1 **do**
  SiftDown(A, i, n);
// Now A is a heap.
for $i \leftarrow n$ **downto** 2 **do**
  swap($A[1], A[i]$)
  SiftDown($A, 1, i-1$)
// Now A is sorted.

Analysis: sorting a heap

SiftDown traverses at most $\log n$ nodes. For each node 2 key comparisons. $\Rightarrow$ sorting a heap costs in the worst case $2 \log n$ comparisons.

Number of memory movements of sorting a heap also $\mathcal{O}(n \log n)$.

Analysis: creating a heap

Calls to siftDown: $n/2$. Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

But mean length of the sift-down paths is much smaller:

$$v(n) = \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{height heaps on level l}} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k$$

$$\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(\mathbf{n})$$

with $s(x) := \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ [10] and $s(\frac{1}{2}) = 2$

[10] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

Intermediate result

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

? Disadvantages of heapsort?

- ! Missing locality: heapsort jumps around in the sorted array (negative cache effect).
- ! Two comparisons required before each necessary memory movement.

8.2 Mergesort

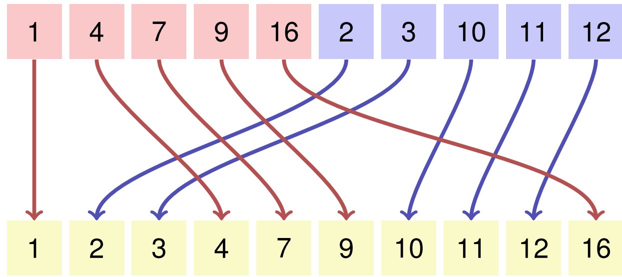
[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Mergesort

Divide and Conquer!

- Assumption: two halves of the array A are already sorted.
- Minimum of A can be evaluated with two comparisons.
- Iteratively: merge the two presorted halves of A in $\mathcal{O}(n)$.

Merge



Algorithm Merge(A, l, m, r)

Input: Array A with length n, indexes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted
Output: $A[l, \dots, r]$ sorted

```
1 B ← new Array(r − l + 1)
2 i ← l; j ← m + 1; k ← 1
3 while i ≤ m and j ≤ r do
4     if A[i] ≤ A[j] then B[k] ← A[i]; i ← i + 1
5     else B[k] ← A[j]; j ← j + 1
6     k ← k + 1;
7 while i ≤ m do B[k] ← A[i]; i ← i + 1; k ← k + 1
8 while j ≤ r do B[k] ← A[j]; j ← j + 1; k ← k + 1
9 for k ← l to r do A[k] ← B[k − l + 1]
```


Correctness

Hypothesis: after k iterations of the loop in line 3 $B[1, \dots, k]$ is sorted and $B[k] \leq A[i]$, if $i \leq m$ and $B[k] \leq A[j]$ if $j \leq r$.

Proof by induction:
Base case: the empty array $B[1, \dots, 0]$ is trivially sorted.
Induction step ($k \to k+1$):

- wlog $A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1, \dots, k]$ is sorted by hypothesis and $B[k] \leq A[i]$.
- After $B[k+1] \leftarrow A[i]$ $B[1, \dots, k+1]$ is sorted.
- $B[k+1] = A[i] \leq A[i+1]$ (if $i+1 \leq m$) and $B[k+1] \leq A[j]$ if $j \leq r$.
- $k \leftarrow k+1, i \leftarrow i+1$: Statement holds again.

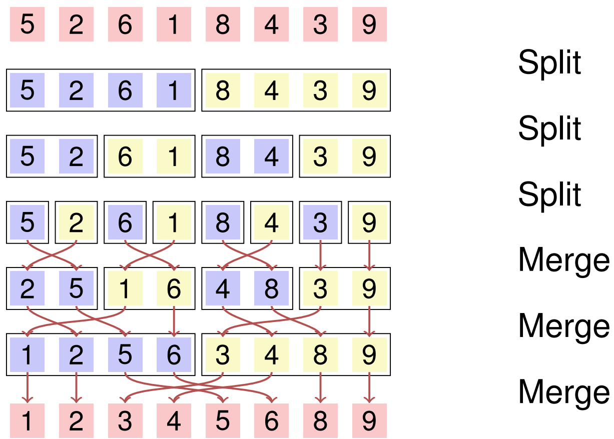
Analysis (Merge)

Lemma

If: array A with length n, indexes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ and $A[l, \dots, m]$, $A[m+1, \dots, r]$ sorted.
Then: in the call of Merge(A, l, m, r) a number of $\Theta(r-l)$ key movements and comparisons are executed.

Proof: straightforward(Inspect the algorithm and count the operations.)

Mergesort



Algorithm (recursive 2-way) Mergesort(A, l, r)

Input: Array A with length n. $1 \leq l \leq r \leq n$
Output: Array $A[l, \dots, r]$ sorted.
if $l < r$ **then**

```
    m ← ⌊(l + r)/2⌋          // middle position
    Mergesort(A, l, m)       // sort lower half
    Mergesort(A, m + 1, r)   // sort higher half
    Merge(A, l, m, r)        // Merge subsequences
```


Analysis

Recursion equation for the number of comparisons and key movements:

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

Algorithm StraightMergesort(A)

Avoid recursion: merge sequences of length $1, 2, 4, ...$ directly

Input: Array A with length n
Output: Array A sorted

$length \leftarrow 1$
while $length < n$ **do** // Iterate over lengths n
  $r \leftarrow 0$
  while $r + length < n$ **do** // Iterate over subsequences
    $l \leftarrow r + 1$
    $m \leftarrow l + length - 1$
    $r \leftarrow \min(m + length, n)$
    Merge(A, l, m, r)
  $length \leftarrow length \cdot 2$

Analysis

Like the recursive variant, the straight 2-way mergesort always executes a number of $\Theta(n \log n)$ key comparisons and key movements.

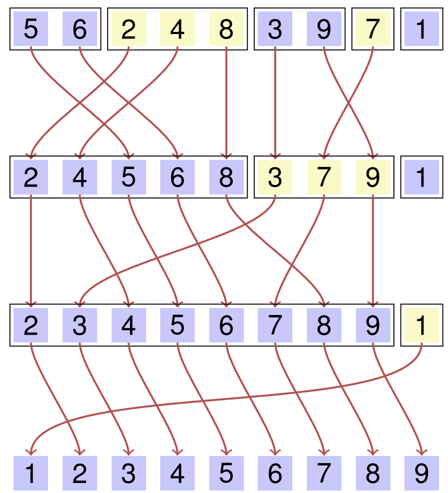
Natural 2-way mergesort

Observation: the variants above do not make use of any presorting and always execute $\Theta(n \log n)$ memory movements.

? How can partially presorted arrays be sorted better?

! Recursive merging of previously sorted parts (*runs) of A.*

Natural 2-way mergesort



Algorithm NaturalMergesort(A)

Input: Array A with length $n > 0$
Output: Array A sorted

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m+1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r+1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```


Analysis

? Is it also asymptotically better than StraightMergesort on average?

! No. Given the assumption of pairwise distinct keys, on average there are $n/2$ positions i with $k_i > k_{i+1}$, i.e. $n/2$ runs. Only one iteration is saved on average.

Natural mergesort executes in the worst case and on average a number of $\Theta(n \log n)$ comparisons and memory movements.

8.3 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Quicksort

? What is the disadvantage of Mergesort?

! Requires additional $\Theta(n)$ storage for merging.

? How could we reduce the merge costs?

! Make sure that the left part contains only smaller elements than the right part.

? How?

! Pivot and Partition!

Use a pivot

1. Choose a (an arbitrary) *pivot* p
2. Partition A in two parts, one part L with the elements with $A[i] \le p$ and another part R with $A[i] > p$
3. Quicksort: Recursion on parts L and R

$\le$	$\le$	$\le$	$\le$	$\le$	p	$>$	$>$	$>$	$>$
1					r				n

Algorithm Partition($A[l..r], p$)

Input: Array A, that contains the pivot p in the interval $[l, r]$ at least once.
Output: Array A partitioned in $[l..r]$ around p. Returns position of p.

```
while l ≤ r do
    while A[l] < p do
        l ← l + 1
    while A[r] > p do
        r ← r − 1
    swap(A[l], A[r])
    if A[l] = A[r] then
        l ← l + 1
return l-1
```


Algorithm Quicksort($A[l, \dots, r]$

Input: Array A with length n. $1 \le l \le r \le n$.
Output: Array A, sorted between l and r.

```
if l < r then
    Choose pivot p ∈ A[l, ..., r]
    k ← Partition(A[l, ..., r], p)
    Quicksort(A[l, ..., k − 1])
    Quicksort(A[k + 1, ..., r])
```


Quicksort (arbitrary pivot)

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

Analysis: number comparisons

Worst case. Pivot = min or max; number comparisons:

$$T(n) = T(n-1) + c \cdot n,\ T(1) = 0 \quad \Rightarrow \quad T(n) \in \Theta(n^2)$$

Analysis: number swaps

Result of a call to partition (pivot 3):

2 1 3 6 8 5 7 9 4

? How many swaps have taken place?

! 2. The maximum number of swaps is given by the number of keys in the smaller part.

Analysis: number swaps

Thought experiment

- Each key from the smaller part pays a coin when it is being swapped.
- After a key has paid a coin the domain containing the key decreases to half its previous size.
- Every key needs to pay at most $\log n$ coins. But there are only n keys.

Consequence: there are $\mathcal{O}(n \log n)$ key swaps in the worst case.

Randomized Quicksort

Despite the worst case running time of $\Theta(n^2)$, quicksort is used practically very often.

Reason: quadratic running time unlikely provided that the choice of the pivot and the pre-sorting are not very disadvantageous.

Avoidance: randomly choose pivot. Draw uniformly from $[l, r]$.

Analysis (randomized quicksort)

Expected number of compared keys with input length n:

$$T(n) = (n-1) + \frac{1}{n}\sum_{k=1}^{n}\left(T(k-1) + T(n-k)\right),\ T(0) = T(1) = 0$$

Claim $T(n) \leq 4n\log n$.

Proof by induction:
Base case straightforward for $n=0$ (with $0\log 0 := 0$) and for $n=1$.
Hypothesis: $T(n) \leq 4n\log n$ for some n.
Induction step: $(n-1 \to n)$

Analysis (randomized quicksort)

$$\begin{aligned}
T(n) &= n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \overset{\text{H}}{\leq} n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} 4k\log k \\
&= n - 1 + \sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n - 1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
&\leq n - 1 + \frac{8}{n}\left((\log n - 1)\sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k\right) \\
&= n - 1 + \frac{8}{n}\left((\log n)\cdot\frac{n(n-1)}{2} - \frac{n}{4}\left(\frac{n}{2}+1\right)\right) \\
&= 4n\log n - 4\log n - 3 \leq 4n\log n
\end{aligned}$$

Analysis (randomized quicksort)

Theorem

On average randomized quicksort requires $\mathcal{O}(n \cdot \log n)$ comparisons.

Practical Considerations

Worst case recursion depth $n-1$[11]. Then also a memory consumption of $\mathcal{O}(n)$.

Can be avoided: recursion only on the smaller part. Then guaranteed $\mathcal{O}(\log n)$ worst case recursion depth and memory consumption.

[11] stack overflow possible!

Quicksort with logarithmic memory consumption

Input: Array A with length n. $1 \leq l \leq r \leq n$.
Output: Array A, sorted between l and r.

while $l < r$ **do**
    Choose pivot $p \in A[l, \dots, r]$
    $k \leftarrow$ Partition$(A[l, \dots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \dots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \dots, r])$
        $r \leftarrow k - 1$

The call of Quicksort($A[l, \dots, r]$) in the original algorithm has moved to iteration (tail recursion!): the if-statement became a while-statement.

Practical Considerations.

- Practically the pivot is often the median of three elements. For example: Median3($A[l], A[r], A[\lfloor l + r/2 \rfloor]$).
- There is a variant of quicksort that requires only constant storage. Idea: store the old pivot at the position of the new pivot.
- Complex divide-and-conquer algorithms often use a trivial ($\Theta(n^2)$) algorithm as base case to deal with small problem sizes.

8.4 Appendix

Derivation of some mathematical formulas

$\log n! \in \Theta(n \log n)$

$$\log n! = \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n$$

$$\begin{aligned}
\sum_{i=1}^{n} \log i &= \sum_{i=1}^{\lfloor n/2 \rfloor} \log i + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log i \\
&\geq \sum_{i=2}^{\lfloor n/2 \rfloor} \log 2 + \sum_{\lfloor n/2 \rfloor + 1}^{n} \log \frac{n}{2} \\
&= (\underbrace{\lfloor n/2 \rfloor}_{> n/2 - 1} - 2 + 1) + (\underbrace{n - \lfloor n/2 \rfloor}_{\geq n/2})(\log n - 1) \\
&> \frac{n}{2} \log n - 2.
\end{aligned}$$

[$n! \in o(n^n)$]

$$n \log n \geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log 2i + \sum_{i=\lfloor n/2 \rfloor+1}^{n} \log i$$
$$= \sum_{i=1}^{n} \log i + \left\lfloor \frac{n}{2} \right\rfloor \log 2$$
$$> \sum_{i=1}^{n} \log i + n/2 - 1 = \log n! + n/2 - 1$$

$$n^n = 2^{n \log_2 n} \geq 2^{\log_2 n!} \cdot 2^{n/2} \cdot 2^{-1} = n! \cdot 2^{n/2-1}$$
$$\Rightarrow \frac{n!}{n^n} \leq 2^{-n/2+1} \overset{n\to\infty}{\longrightarrow} 0 \Rightarrow n! \in o(n^n) = \mathcal{O}(n^n) \backslash \Omega(n^n)$$

[Even $n! \in o((n/c)^n)\, \forall\, 0 < c < e$]

Konvergenz oder Divergenz von $f_n = \frac{n!}{(n/c)^n}$.

Ratio Test

$$\frac{f_{n+1}}{f_n} = \frac{(n+1)!}{\left(\frac{n+1}{c}\right)^{n+1}} \cdot \frac{\left(\frac{n}{c}\right)^n}{n!} = c \cdot \left(\frac{n}{n+1}\right)^n \longrightarrow c \cdot \frac{1}{e} \lessgtr 1 \text{ if } c \lessgtr e$$

because $\left(1 + \frac{1}{n}\right)^n \to e$. Even the series $\sum_{i=1}^{n} f_n$ converges / diverges for $c \lessgtr e$.

f_n diverges for $c = e$, because (Stirling): $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$.

[Ratio Test]

Ratio test for a sequence $(f_n)_{n\in\mathbb{N}}$: If $\frac{f_{n+1}}{f_n} \underset{n\to\infty}{\longrightarrow} \lambda$, then the sequence f_n and the series $\sum_{i=1}^{n} f_i$

- converge, if $\lambda < 1$ and
- diverge, if $\lambda > 1$.

[Ratio Test Derivation]

Ratio test is implied by Geometric Series

$$S_n(r) := \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}.$$

converges for $n \to \infty$ if and only if $-1 < r < 1$.

Let $0 \leq \lambda < 1$:

$$\forall \varepsilon > 0\, \exists n_0 : f_{n+1}/f_n < \lambda + \varepsilon\, \forall n \geq n_0$$
$$\Rightarrow \exists \varepsilon > 0, \exists n_0 : f_{n+1}/f_n \leq \mu < 1\, \forall n \geq n_0$$

Thus

$$\sum_{n=n_0}^{\infty} f_n \leq f_{n_0} \cdot \sum_{n=n_0}^{\infty} \cdot \mu^{n-n_0} \quad \text{konvergiert.}$$

(Analogously for divergence)