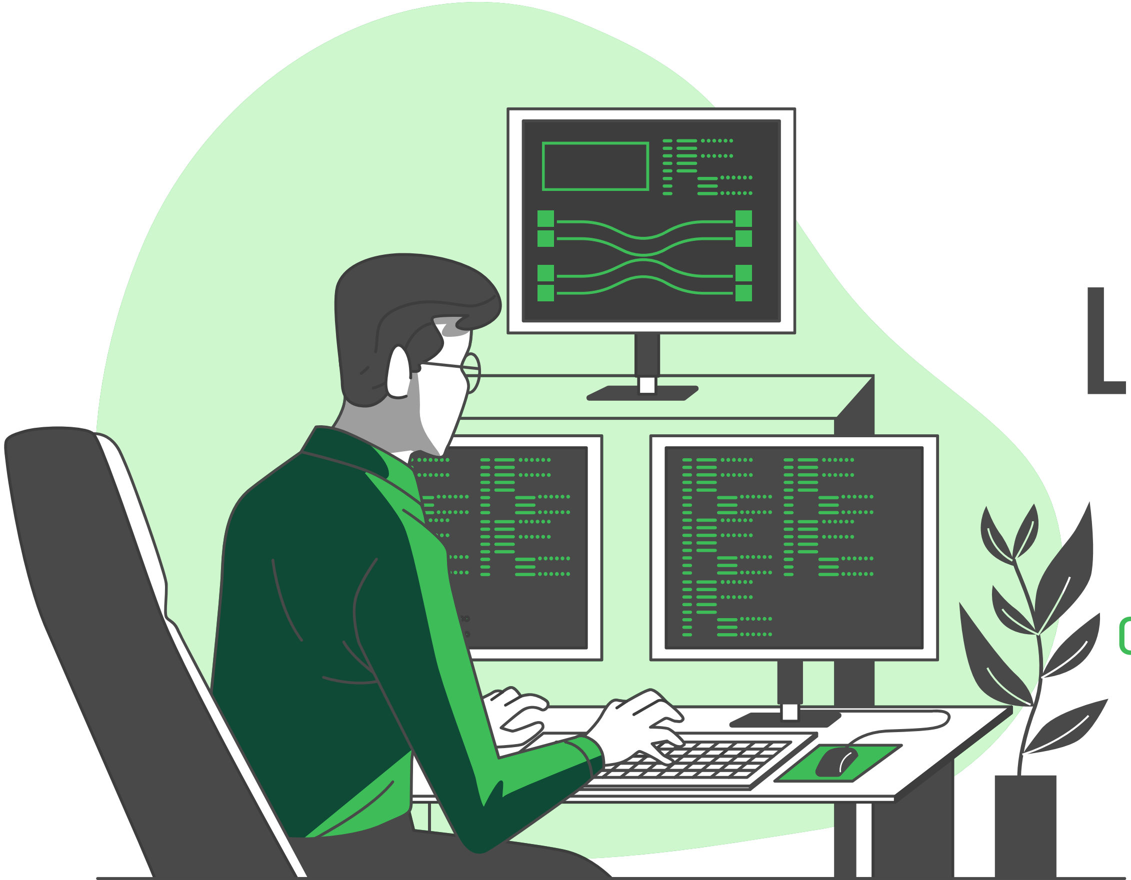


Linearithmic Sorting

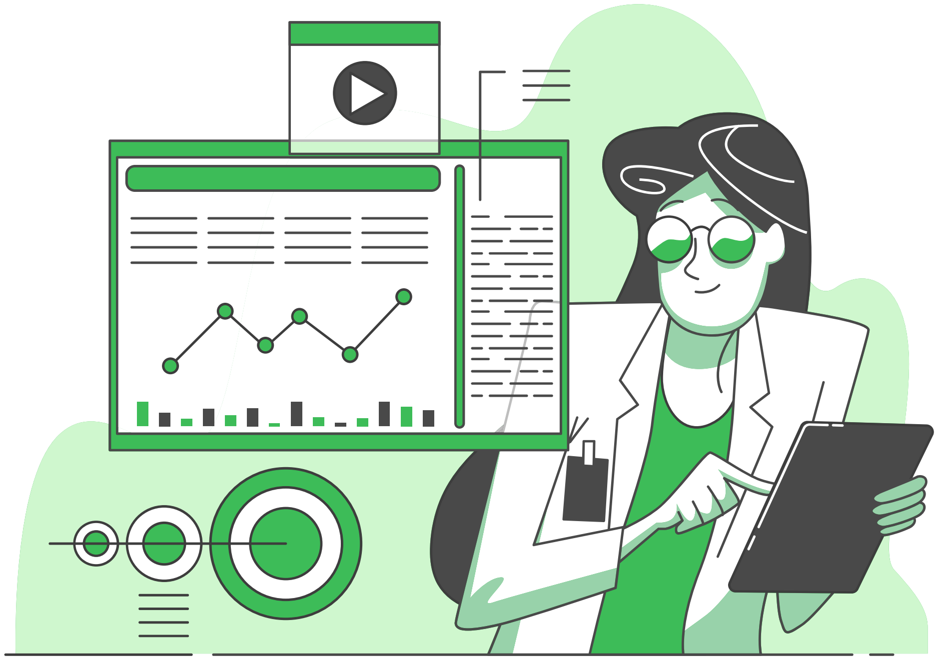
CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

Table of Contents

01
Heap Sort
Sorting using binary heaps

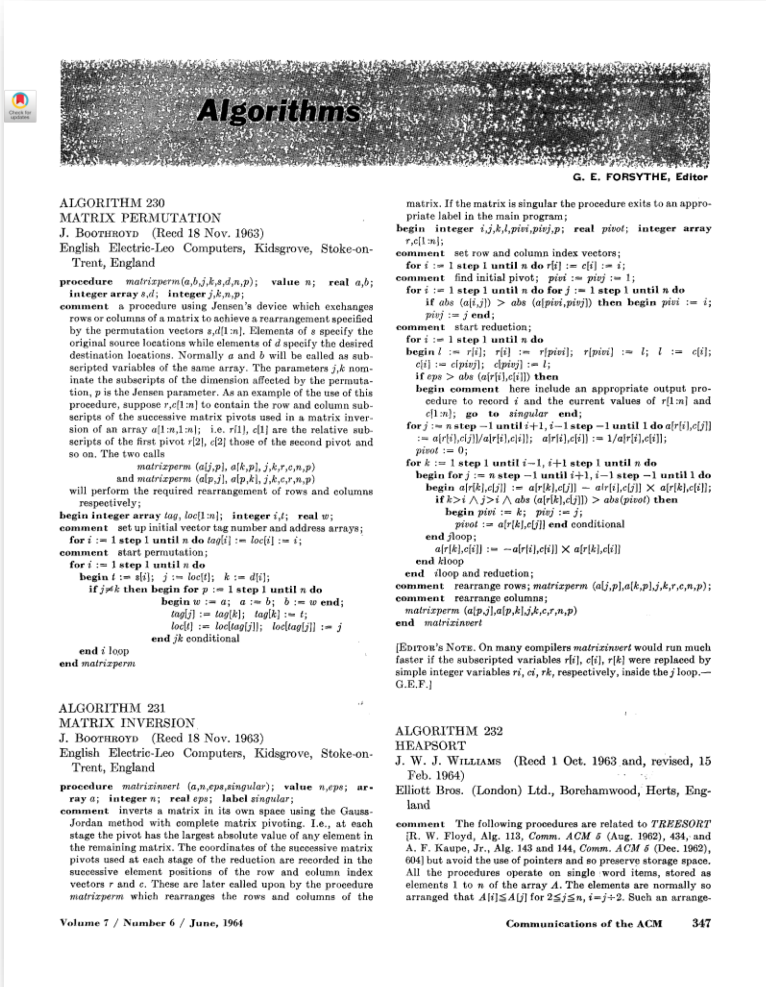
02
Quick Sort
Sorting using pivoted partitions



01
Heap Sort

Sorting using binary heaps

"Algorithm 232 – Heapsort", J. W. J. Williams, "Communications of the ACM", 1964



Algorithms

G. E. FORSYTHE, Editor

ALGORITHM 230
MATRIX PERMUTATION
J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixperm(a,b,j,k,s,d,n,p); value n; real a,b;
  integer array s,d; integer j,k,n,p;
comment a procedure using Jensen's device which exchanges
  rows or columns of a matrix to achieve a rearrangement specified
  by the permutation vectors s,d[1:n]. Elements of s specify the
  original source locations while elements of d specify the desired
  destination locations. Normally a and b will be called as sub-
  scripted variables of the same array. The parameters j,k nom-
  inate the subscripts of the dimension affected by the permuta-
  tion, p is the Jensen parameter. As an example of the use of this
  procedure, suppose r,c[1:n] to contain the row and column sub-
  scripts of the successive matrix pivots used in a matrix inver-
  sion of an array a[1:n,1:n]; i.e. r[1], c[1] are the relative sub-
  scripts of the first pivot r[2], c[2] those of the second pivot and
  so on. The two calls
                  matrixperm (a[j,p], a[k,p], j,k,r,c,n,p)
              and matrixperm (a[p,j], a[p,k], j,k,c,r,n,p)
  will perform the required rearrangement of rows and columns
    respectively;
begin integer array tag, loc[1:n]; integer i,t; real w;
comment set up initial vector tag number and address arrays;
  for i := 1 step 1 until n do tag[i] := loc[i] := i;
comment start permutation;
  for i := 1 step 1 until n do
    begin t := s[i]; j := loc[t]; k := d[i];
      if j≠k then begin for p := 1 step 1 until n do
                        begin w := a; a := b; b := w end;
                          tag[j] := tag[k]; tag[k] := t;
                          loc[t] := loc[tag[j]]; loc[tag[j]] := j
                      end jk conditional
    end i loop
end matrixperm
```

ALGORITHM 231
MATRIX INVERSION
J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixinvert (a,n,eps,singular); value n,eps; ar-
  ray a; integer n; real eps; label singular;
comment inverts a matrix in its own space using the Gauss-
  Jordan method with complete matrix pivoting. I.e., at each
  stage the pivot has the largest absolute value of any element in
  the remaining matrix. The coordinates of the successive matrix
  pivots used at each stage of the reduction are recorded in the
  successive element positions of the row and column index
  vectors r and c. These are later called upon by the procedure
  matrixperm which rearranges the rows and columns of the
  matrix. If the matrix is singular the procedure exits to an appro-
  priate label in the main program;
begin integer i,j,k,l,pivi,pivj,p; real pivot; integer array
  r,c[1:n];
comment set row and column index vectors;
  for i := 1 step 1 until n do r[i] := c[i] := i;
comment find initial pivot; pivi := pivj := 1;
  for i := 1 step 1 until n do for j := 1 step 1 until n do
      if abs (a[i,j]) > abs (a[pivi,pivj]) then begin pivi := i;
      pivj := j end;
comment start reduction;
  for i := 1 step 1 until n do
  begin l := r[i]; r[i] := r[pivi]; r[pivi] := l; l := c[i];
    c[i] := c[pivj]; c[pivj] := l;
    if eps > abs (a[r[i],c[i]]) then
    begin comment here include an appropriate output pro-
      cedure to record i and the current values of r[1:n] and
      c[1:n]; go to singular end;
  for j := n step -1 until i+1, i-1 step -1 until 1 do a[r[i],c[j]]
    := a[r[i],c[j]]/a[r[i],c[i]]; a[r[i],c[i]] := 1/a[r[i],c[i]];
    pivot := 0;
  for k := 1 step 1 until i-1, i+1 step 1 until n do
    begin for j := n step -1 until i+1, i-1 step -1 until 1 do
      begin a[r[k],c[j]] := a[r[k],c[j]] - a[r[i],c[j]] × a[r[k],c[i]];
        if k>i ∧ j>i ∧ abs (a[r[k],c[j]]) > abs(pivot) then
          begin pivi := k; pivj := j;
            pivot := a[r[k],c[j]] end conditional
      end jloop;
        a[r[k],c[i]] := -a[r[i],c[i]] × a[r[k],c[i]]
    end kloop
  end iloop and reduction;
comment rearrange rows; matrixperm (a[j,p],a[k,p],j,k,r,c,n,p);
comment rearrange columns;
  matrixperm (a[p,j],a[p,k],j,k,c,r,n,p)
end matrixinvert
```

[Editor's Note. On many compilers *matrixinvert* would run much faster if the subscripted variables $r[i]$, $c[i]$, $r[k]$ were replaced by simple integer variables ri, ci, rk, respectively, inside the j loop.—G.E.F.]

ALGORITHM 232
HEAPSORT
J. W. J. Williams (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)
Elliott Bros. (London) Ltd., Borehamwood, Herts, England

```
comment The following procedures are related to TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434, and
  A. F. Kaupe, Jr., Alg. 143 and 144, Comm. ACM 5 (Dec. 1962),
  604] but avoid the use of pointers and so preserve storage space.
  All the procedures operate on single word items, stored as
  elements 1 to n of the array A. The elements are normally so
  arranged that A[i]≤A[j] for 2≤j≤n, i=j÷2. Such an arrange-
```


https://dl.acm.org/doi/10.1145/512274.512284

Heap Sort Algorithm

1. Build a **binary heap** and sort down values.

2. heapify: Transforms the input array A into a binary heap (in-place).

3. Which heap order?
   a. **Max-Heap** if sorting in non-descending order.
   b. **Min-Heap** if sorting in non-ascending order.

4. Sort down: Move Max/Min value to the end of the array. Readjust the heap, and repeat.

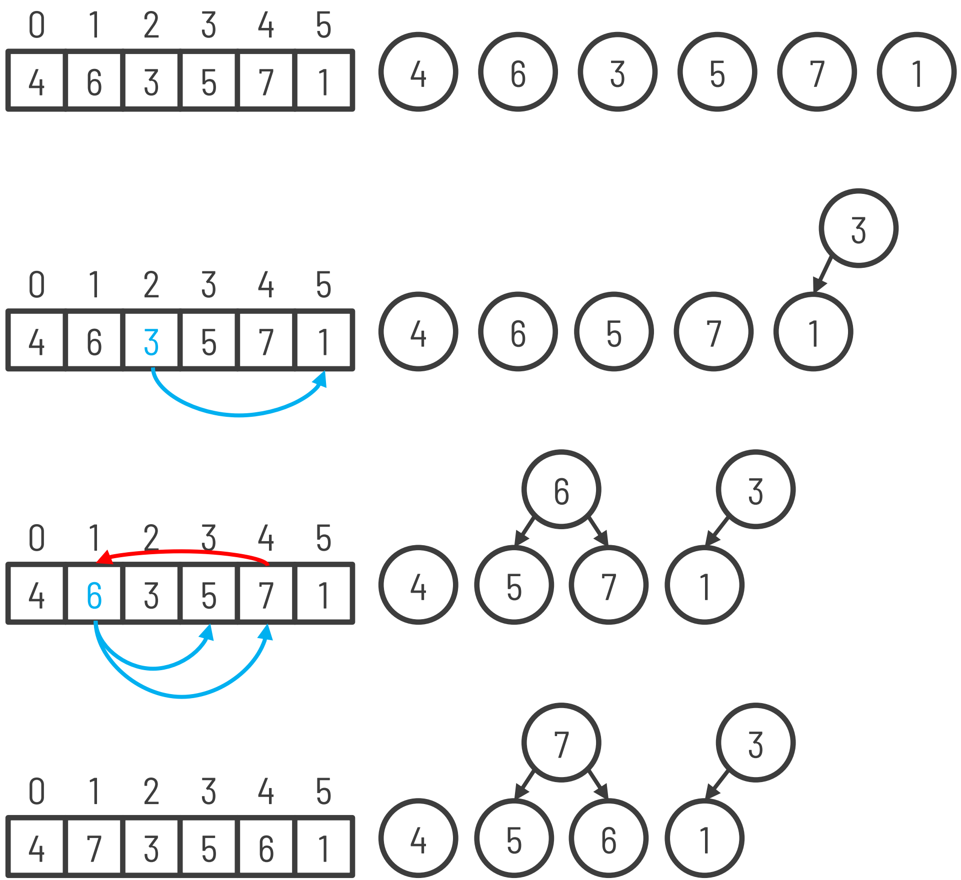
Heapify and Heap Sort

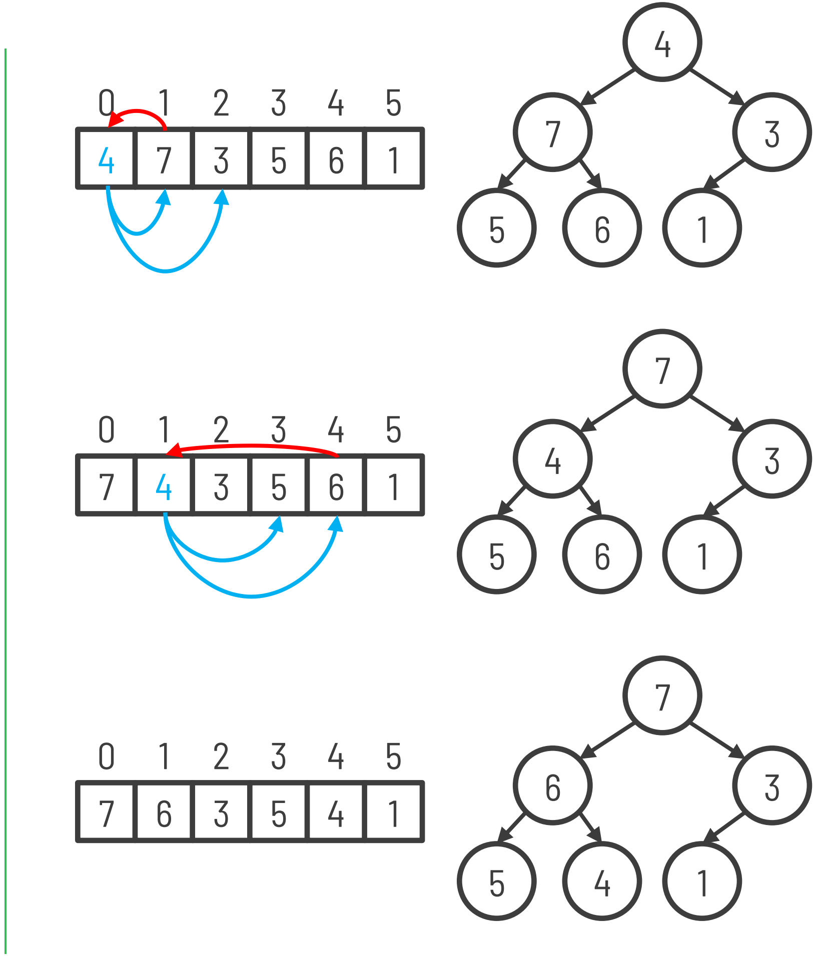
```
algorith heapify(A:array, n:ℤ≥0)
   for i from floor(n/2) – 1 to 0 by -1 do
      siftdown(A, n, i)
   end for
end algorithm
```

```
algorith heapsort(A:array)
   let n be the size of A
   heapify(A, n)
   for i from n-1 to 1 by -1 do
      swap(A, 0, i)
      n ← n - 1
      siftdown(A, n, 0)
   end for
end algorithm
```

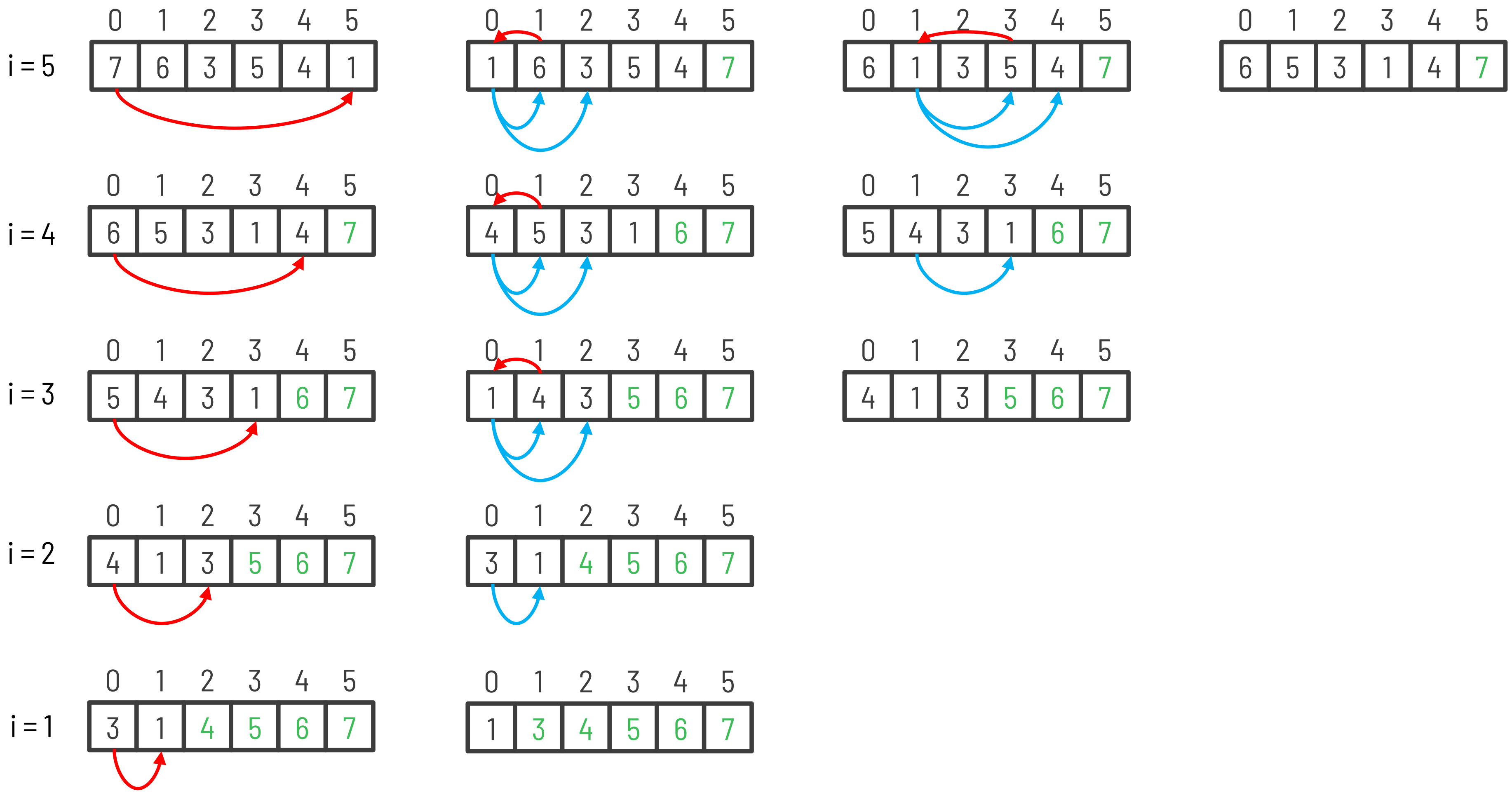
Build heap: [4, 6, 3, 5, 7, 1]

heapify (transform an array into a binary heap)





Sortdown: Swap the root with the last item of the binary heap. Then, fix the “new” binary heap.



Sort Down Remarks

- Sorting in ascending order? Exchange maximum value with the last element of the array.
- Reorder the rest of the heap. Do not mess with elements at the end of the array. Remember why?
- Runtime: call sink $n-1$ times, each one is $O(\log_2(n))$. So, $O(n\log_2(n))$.
- Heap Sort runtime: $T(n) \approx O(n) + O(n\log_2(n)) \in O(n\log_2(n))$.

Heap Sort Remarks

The good:

a. Runtime: $\Theta(n \log(n))$. Remember what it means?

b. Space: $\Theta(1)$ (aka. In-place).

c. Easy to implement. Very short!

The bad:

a. Not that good in practice due to cache issues (pay attention to CS250 and CS354)

02
Quick Sort

Sorting using pivoted partitions

Recall Merge Sort

```
algorithm mergesort(A:array, l:ℤ≥0, r:ℤ≥0)
   if l < r then
      m ← floor((l+r) / 2)
      mergesort(A, l, m)
      mergesort(A, m + 1, r)
      merge(A, l, m, r)
   end if
end algorithm
```

First call:

```
let A be an array with n comparable items
mergesort(A, 0, n-1)
```

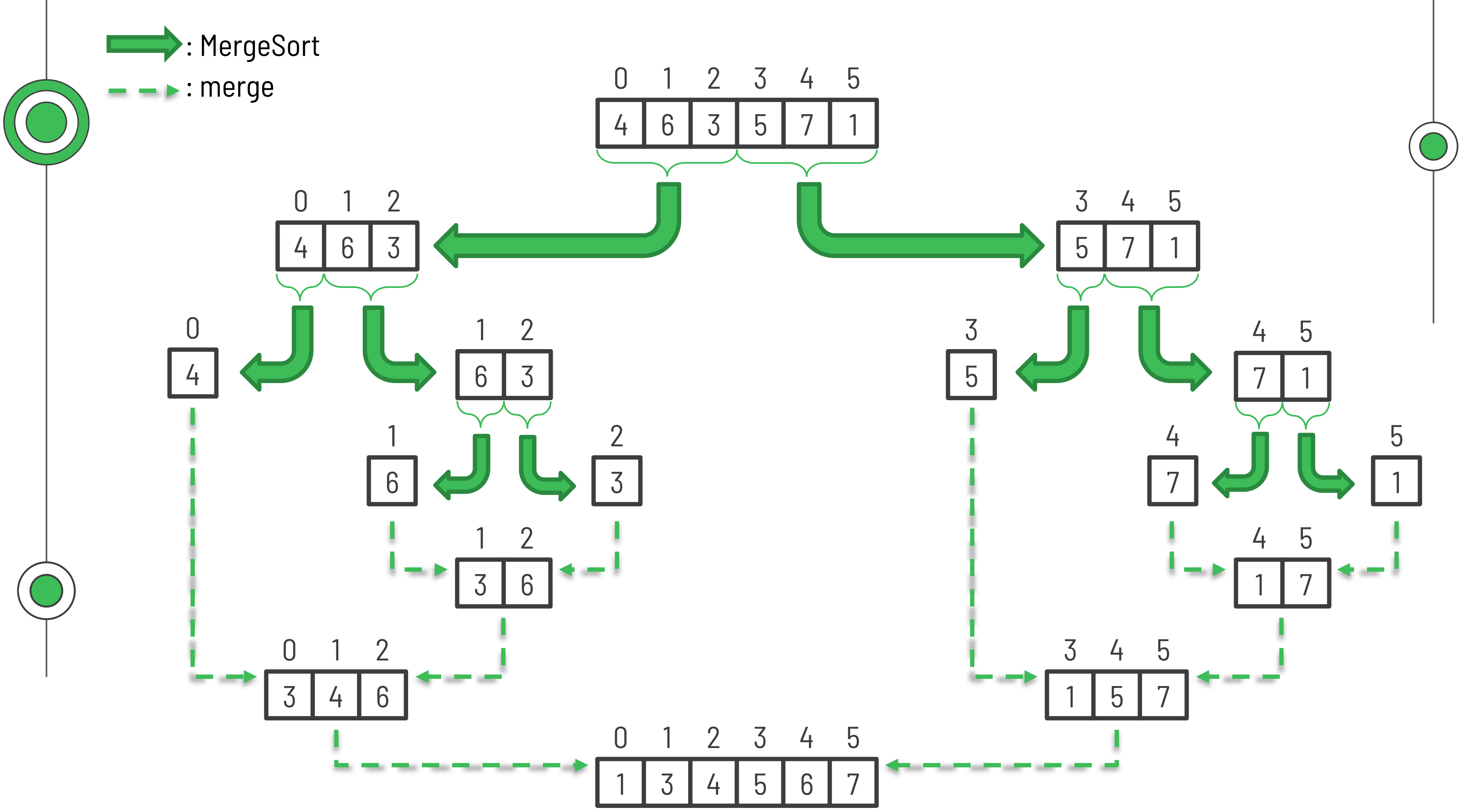
```
algorithm merge(A:array, l:ℤ≥0, m:ℤ≥0, r:ℤ≥0)
   n1 ← m – l + 1
   n2 ← r – m
   let L be an array of size n1 + 1
   let R be an array of size n2 + 1

   for i from 0 to n1 – 1 do
      L[i] ← A[l + i]
   end for

   for j from 0 to n2 – 1 do
      R[j] ← A[m + j + 1]
   end for

   L[n1] ← ∞, R[n2] ← ∞
   i ← 0, j ← 0

   for k from l to r do
      if L[i] <= R[j] then
         A[k] ← L[i]
         i ← i + 1
      else
         A[k] ← R[j]
         j ← j + 1
      end if
   end for
end algorithm
```



: MergeSort
: merge
0 1 2 3 4 5
4 6 3 5 7 1
0 1 2
4 6 3
3 4 5
5 7 1
0
4
1 2
6 3
3
5
4 5
7 1
1
6
2
3
4
7
5
1
1 2
3 6
4 5
1 7
0 1 2
3 4 6
3 4 5
1 5 7
0 1 2 3 4 5
1 3 4 5 6 7

Merge Sort Remarks

- Algorithm paradigm: Divide & Conquer
- Recursive: $T(n) = 2T\left(\frac{n}{2}\right) + c_r n$, $T(1) = c_b$
- Non-recursive: $T(n) = c_r n \log_2(n) + c_b n$
- Runtime complexity: $T(n) \in \Theta(n \log_2(n))$
- Space: $\Theta(n)$ (**not** an **in-place** sorting algorithm)
- A popular among software developers and tech interviewers.

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. Communications of the ACM, 4(7), 321.

ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  quicksort (A,M,N);  value M,N;
             array A;  integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin        integer I,J;
             if M < N then begin partition (A,M,N,I,J);
                                  quicksort (A,M,J);
                                  quicksort (A, I, N)
                            end
end          quicksort
```

ALGORITHM 65
FIND
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  find (A,M,N,K);  value M,N,K;
             array A;  integer M,N,K;
comment  Find will assign to A [K] the value which it would
have if the array A [M:N] had been sorted. The array A will be
partly sorted, and subsequent entries will be faster than the first;
```


https://dl.acm.org/doi/10.1145/366622.366644

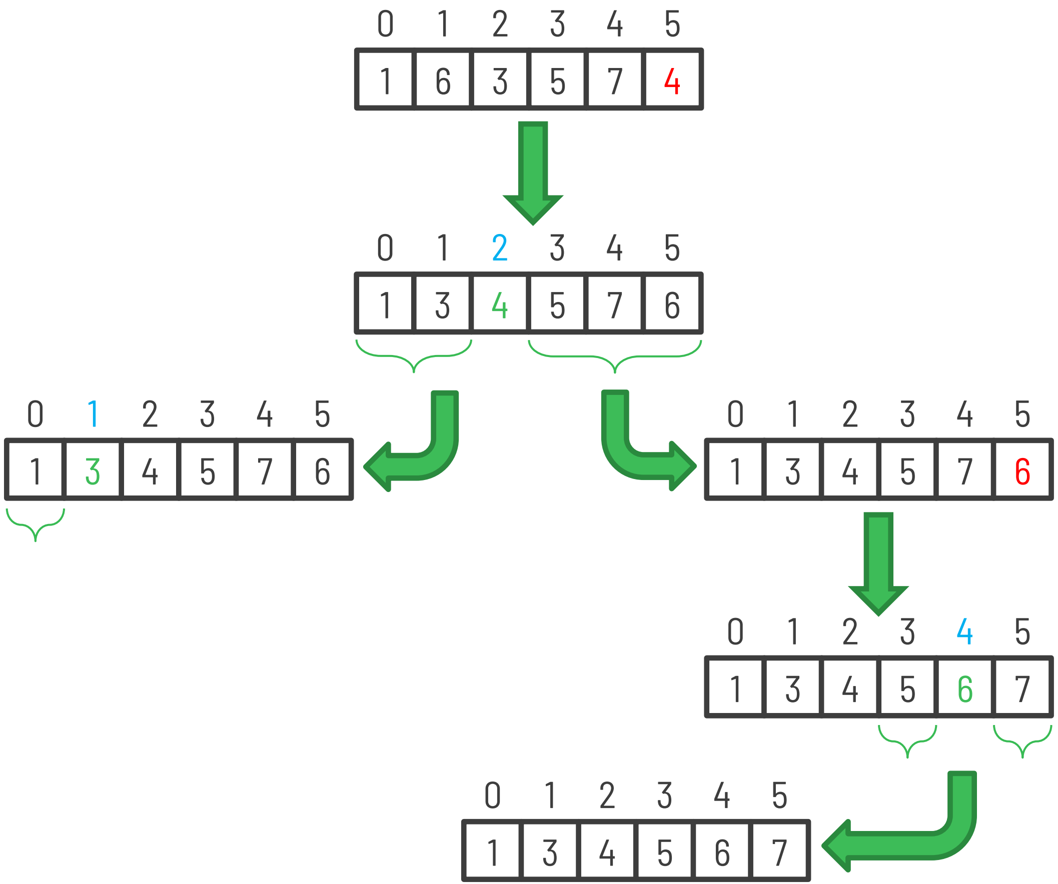
Quick Sort Algorithm

```
algorithm quicksort(A:array, l:ℤ≥0, r:ℤ≥0)
   if l < r then
      m ← partition(A, l, r)
      quicksort(A, l, m - 1)
      quicksort(A, m + 1, r)
   end if
end algorithm
```

First call:

```
let A be an array with n comparable items
quicksort(A, 0, n-1)
```

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l − 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```


0 1 2 3 4 5
1 6 3 5 7 4
0 1 2 3 4 5
1 3 4 5 7 6
0 1 2 3 4 5
1 3 4 5 7 6
0 1 2 3 4 5
1 3 4 5 7 6
0 1 2 3 4 5
1 3 4 5 6 7
0 1 2 3 4 5
1 3 4 5 6 7

Randomized Quick Sort Algorithm

```
algorithm quicksort(A:array, l:ℤ≥0, r:ℤ≥0)
   if l < r then
      m ← randpartition(A, l, r)
      quicksort(A, l, m - 1)
      quicksort(A, m + 1, r)
   end if
end algorithm
```

First call:

```
let A be an array with n comparable items
quicksort(A, 0, n-1)
```

```
algorithm randpartition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   i ← randominteger(l, r)
   swap(A, i, r)
   return partition(A, l, r)
end algorithm
```

```
algorithm partition(A:array, l:ℤ≥0, r:ℤ≥0) → ℤ≥0
   p ← A[r]
   i ← l − 1
   for j from l to r - 1 do
      if A[j] < p then
         i ← i + 1
         swap(A, i, j)
      end if
   end for
   i ← i + 1
   swap(A, i, r)
   return i
end algorithm
```

Quick Sort is not Stable

Insight: Given an array A of n elements with keys $a_0, a_1, \dots, a_{n-1}$, quick sort selects a "pivot" and partitions the array into two subarrays: one with elements less than the pivot and the other with elements greater than the pivot. The **choice of pivot** and the **method of partitioning** can cause elements with equal keys to be reordered.

The pivot selection is **arbitrary**, and elements are rearranged based on their comparison with the pivot. This rearrangement does not necessarily preserve the original order of elements with equal keys.

When elements are moved during partitioning, their original relative positioning is not guaranteed to be maintained unless additional measures are taken.

Quick Sort Analysis

Reality: Not a trivial analysis due to the randomness while selecting a pivot and building the partitions.

Best case:

a. Pivot value to be the median value of the array, making the partitions to be divided evenly (**balanced partitions**).
b. Time complexity like Merge Sort: $O(n \log_2(n))$.

Worst case:

a. Pivot to be the minimum or maximum value in the array. Making one of the partitions to have $n - 1$ elements (**unbalanced partitions**).
b. Time Complexity: $T(n) = n + (n-1) + (n-2) + \cdots 2 + 1 \in O(n^2)$

Big Issue: Having a Good Pivot

Value at some random index? Value at median index? First element? Last element?

It is hard to tell.

So, why do we care about Quick Sort:

- It is fast!
- We can do it in-place.
- Unlikely (but not impossible) to fall into the worst-case.

Done, Sort Of

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik and illustrations by Stories