A1

Algoritmen.

Euclid's GCD

gcd of a,b, gegeven: $a \geq b$
if $b=0$, return a
if $b \neq 0$, return gcd(b, a mod b)

Insertion Sort.

- Idea: sorted part is only first element, while nonsorted is nonempty, insert first element in right place.
- Pseudo code:

```
Algorithm insertionSort (A, n)
  for j:=2 to n do
      key:= A[j]
      i:= j-1
      while i≥1 and A[i]>key do
          A[i+1]:= A[i]
          i:= i-1
      A[i+1]:= key
```

- Proof: Loop invariant: at the start of the for-loop the sub-array A[1, ..., j-1] is a sorted permutation of the sub-array A[1, ..., j-1] of the input-array
  - init: I is initially true for {j=2}
  - loop: I remains true during loop
  - end: I gives correctness for j=n+1.
- worst case complexity: $\Theta(n^2)$

arrayMax

(input: Array of n integers, output: max)

- PSEUDO code:

```
currentMax:= A[1]
for i:=2 to n do
```

~~if~~ if currentMax < ~~A~~ A[i] then
    current max := A[i]
  return currentMax

- worst-case time complexity in $O(n)$
  best-case time complexity in $\Omega(n)$, so $\Theta(n)$

Selection Sort.

- Idea: initially sorted part is empty. look for smallest, swap with first of unsorted.
- Pseudo code

```
Algorithm SelectionSort(A,n)
  for i:=1 to n-1 do
     m:i
     for j=i+1 to n do
        if A[j] < A[m] then
           m:=j
     x:= A[m]     }
     A[m]:=A[i]   } Swap (A[m], A[i])
     A[i]:=x      }
```

- Worst case time complexity: $\Theta(n^2)$

Bubble Sort

- Idea: Repeatly go from left to right, compare two neighbours, swap if in wrong order.
- Pseudo code

```
Algorithm BubbleSort(A,n)
  for i:=n-1 downto 1 do
     for j:=1 to i do
        if A[j] > A[j+1] then
           swap (A[j], A[j+1])
```

Merge Sort
- idea: split, sort subseq. recursively, merge
  idea merge: compare first elements, remove smallest, → end of outp
- Pseudo-code merge Sort.

```
Algorithm mergeSort(A, p, r)
  if p<r then
    q := ⌊(p+r)/2⌋
    mergeSort (A, p, q)
    mergeSort (A, q+1, r)
    merge (A, p, q, r)
```

- worstcase time complexity: $\Theta(n \log n)$
- Not in-place!

MaxSubArray (in $\Theta(n^2)$)
- Algorithm maxSubArray(A, n)

```
  max := 0
  for left := 1 to n do
    sum := 0
    for right := left to n do
      sum := sum + A[right]
      if sum > max then
        max := sum
  return max
```

- can be done in $\Theta(n \log n)$ with divide and conquer. (array is either left, right or using middle

~~Down~~ downMaxHeap (A, i)
- idea: We have a node with left and right heaps. reconstruct heap properly using downheap
- Algorithm: zoz.

Algorithm ~~6~~ downMaxHeap (A,i)

```
l := left(i)
r := right(i)
if l ≤ A.heap-size and A[l] > A[i] then
    largest := l
else
    largest := i
if r ≤ A.heap-size and A[r] > A[largest] then
    largest := r
if largest ≠ i then
    swap(A[i], A[largest])
    downMaxHeap(A, largest)
```

- time complexity: $O(\log(n))$

Build ~~m~~ MaxHeap(H):

- assume node has binary tree at its ~~roots~~ roots, ~~co~~ perform downMaxHeap
- Algorithm buildMaxHeap(H):

```
H.size := H.length
for i = ⌊H.length/2⌋ downto 1 do
    downMaxHeap(H,i)
```

- You would assume: ~~Θ~~ $O(n \log n)$, but a subtle analysis gives $O(n)$*

*for every height j in $0, \dots, \lfloor \log n \rfloor$, for each of the at most $\lceil \frac{n}{2^{j+1}} \rceil$ nodes of height j we do downMaxHeap which is in $O(j)$, so bMH in $O(n)$

HeapSort.

- Idea: turn array into max-heap, swap root with key on last node, exclude last node and reconstruct the heap.
- Algorithm heapSort(H)

```
buildMaxHeap(H)
for i = H.length downto 2 do
```

swap H[i] and H[i]
H.heap-size: = H.heap-size -1
downMaxHeap (H,1)

- ~~Build~~ running time in O(nlogn)

~~...~~ 3 Methods for priority queues:

1 -Algorithm maximum (H) //returns. does not remove
return H[1] (in O(1))

2 -Algorithm extractMax (H) // returns & removes
max:= H[1]
H[1]:= H[H.heap-size]
H.heapsize -= 1;
downMaxHeap (H,1)
return Max

- Running time in O(log(n)

3 -Algorithm insert (H, k)
H.heapsize += 1
H[heap-size] = k;
upMaxHeap(H, H.heap-size (in O(logn))

Bubbling up in max heap:

- Algorithm upMaxHeap (H,i)
~~A~~ while i>0 and H[parent(i)] < H[i] do
swap (H[parent(i)], H[i])
i:= parent (i).

- Algorithm does: we have a heap and add a labelled node, we reconstruct the heap property

QuickSort

- Idea: divide into two parts, up to index q containing keys less than ~~pivot~~ or equal to pivot. Recur sort ~~r~~ecursively the two arrays conquer combine the two solutions
- Algorithm, to sort $A[p....r]$

```
Algorithm quickSort(A,p,r)
  if p<r then
    q:= partition(A,p,r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)
```

Part of Quicksort: partition

- Idea: while list contains > 1 element, take pivot k from list. Index i: last key of small ones so far. Index j: first key to be compared with pivot. If j finds a key smaller than pivot, swap that key with the one at i+1.
- Algorithm partition(A,p,r)

```
  x:= A[r]
  i:= p-1
  for j=p to r-1
    if A[j] ≤ x
      i := i+1
      exchange A[i] and A[j]
  exchange A[i+1] and A[r]
  return i+1
```

- Correctness:
  - for k: if $p \leq k \leq i \Rightarrow A[k] \leq pivot$
  - if $i+1 \leq k \leq j-1 \Rightarrow A[k] > pivot$
  - if $k = r$ then $A[k] = pivot$.

holds initially, through loop and yields correctness

correctness of Quicksort follows from correctness of partition and induction.
Quicksort worst case running time: $\Theta(n^2)$,
best case running time: ~~[illegible]~~ $\Theta(n \log n)$

Randomized Quicksort.
- pivot is random key
- good pivot: small and large part of partition $< \frac{3}{4}$
- probability of good pivot: $1/2$
- Expected running time: $O(n \log n)$

Counting Sort. input from range: $\{0, \dots, k\}$
(non comparison based sorting algorithm)
- Idea: count number of occurances of each i from 0 to k
- time complexity: $\Theta(n+k)$ for an input array length n.
- drawback: fixed range, requires additional array length k.
- Algorithm countingSort(A, B, k) // B: output array.

```
new ~~Array~~ array C[0, ..., k]
for i = 0 to k do            // initialize
  C[i] := 0
for j := 1 to A.length do    // count
  C[A[j]] = C[A[j]] + 1
for i := 1 to k do           // make cumulative
  C[i] = C[i] + C[i-1]
for j := A.length downto 1 do
  B[C[A[j]]] := A[j]
  C[A[j]] := C[A[j]] - 1
```

Radix Sort. (for lexicographic ordering
- for i := 1 to d do
  use stable (=equal keys stay in order) sort on digit d
- if stable sorting alg. is in $\Theta(n+k)$, then radix is in $\Theta(d*(n+k))$

Bucket Sort (non-comparison based)

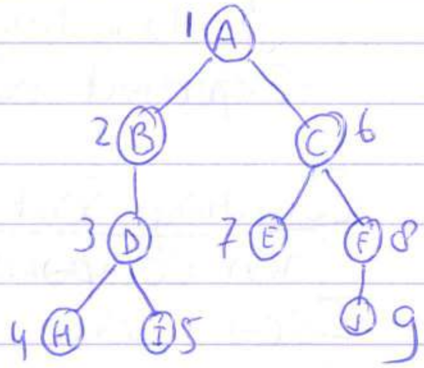
items have label, store under label and join the buckets.

Tree traversals. [How can we visit all nodes in a tree exactly once]

- Preorder traversal (in $O(n)$)

First visit the node and its successors.

```
Algorithm preOrder(v)
  visit(v)
  if v.left ≠ nil then
     preOrder(v.left)
  if v.right ≠ nil then
     preOrder(v.right)
```

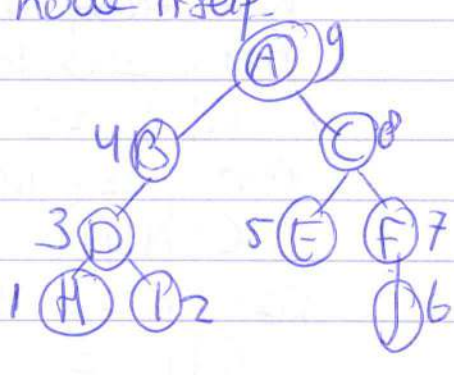


PreOrder

- Postorder traversal

First visit all successors and next node itself.

```
Algorithm postOrder(v)
  if v.left ≠ nill then
     postOrder(v.left)
  if v.right ≠ nill then
     postorder(v.right)
  visit(v)
```



- Inorder traversal

visit left subtree, visit node, visit right sub tree.

- Euler traversal

generic description of traversals

AS $\frac{36}{60}$

~~A1~~

instantiate visitLeft, visitBelow, visitRight as desired

```
Algorithm EulerTour(v):
  visitLeft(v)
  if v.left ≠ nil
    eulerTour(v.left)
  visitBelow(v)
  if v.right ≠ nil
    eulerTour(v.right)
  visitRight(v).
```

Fibonacci Numbers. (naive, memoization, dynamic)

- Naive:

```
Algorithm fib(n)
  if n=1 or n=2 then
    return 1
  else
    f := fib(n-1) + fib(n-2)
  return f
```

Exponential running time: $T(n) = T(n-1) + T(n-2) + c$

- Memoization.

take array $r[1, \dots, n]$ and initialize at zero. $\forall i$

```
Algorithm fib(n)
  if r[n] ≠ 0 then
    return r[n]
  if n=1 or n=2 then
    return 1
  else
    f: fib(n-1) + fib(n-2)
  r(n) = f
  return f.
```

- dynamic
Algorithm fib(n)
  new array [1, ..., n]
  r[1] := 1
  r[2] := 2
  for i := 3 to n do
    ~~fib~~ r[i] = ~~fib~~[i-2] + ~~fib~~[i-1]
  return r[n].

<u>Max Sub Array</u> [naive, divide and conquer, dp]

- Naive:
~~Algorithm maxSubArray(A, n)~~
  ~~max := 0~~
  ~~for left :=~~
Staat eerder in su.

- Divide and conquer
(te) Lang, zie slides.

- DP.
Idea: Create new array B[1, ..., n]
B[r] is maximal sum of subarray ending at index
r. Start: B[1] = max { A[1], 0 }.
At r: either emty subarray ends, or maxsubarray ending at r-1 is continued
So B[r] = max { 0, ~~A~~ B[r-1] + A[r] }
<u>Algorithm</u>: maxSubArray(A, n)
  new Array B
  B[1] := max { 0, A[1] }
  m := B[1]

```
for r=2 to n do
    B[r] := max {0, B[r-1] + A[r]}
    m := max (m, B[r])
return m.
```

Rod cutting (recursive / DP)

Problem: rod of n decimeters. Determine max revenue when price p_i is given for a rod of i decimeters.

There are 2^{n-1} possibilities of cutting.

```
Algorithm rodCuttingRecursive(p,m)
  if m=0
    return 0                    // input: p[1, ..., n] with prices.
  q := -∞
  for i:=1 to m do
    q := max(q, p[i] + rodCuttingRecursive(p, m-i))
  return q.
```

takes $T(n) = 2^n$ time

```
Algorithm rodCuttingDP(p,n)
  new Array b[0, ..., n]
  b[0] := 0
  for j:=1 to n do
    q := -∞
    for i:= 1 to j do
      q := max(q, p[i] + b[j-i])
    b[j] := q
  return b[n]
```

Worst case: $O(n^2)$

Matrix Multiplication. // A: $p \times q$ B: $q \times r$

Algorithm matrixMultiply (A, B)
  new matrix C $p \times r$
  for $i := 1$ to p do
    for $j := 1$ to r do
      $c_{ij} := 0$
    for $k := 1$ to q do
      $c_{ij} = c_{ij} + a_{ik} b_{kj}$

time complexity determined by $p \cdot q \cdot r$

Question: how to put parenthesis in longer multiplication?

So general problem: $A_1 \cdots\cdots A_n$
- dimension of A_i : $d_{i-1} \times d_i$
- idea for $m_{i,j}$: number of steps for opt. parenthesis for $A_i \cdots A_j$
- definition of $m_{i,j}$

$$m_{i,j} = \min_{i \le k \le j} \{ m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j \}$$

$[A_i \cdots A_k$ has dimensions $d_{i-1} \times d_k]$

Algorithm matrixChain(d)
  new table $m[1, \ldots, n; 1, \ldots, n]$ // is matrix
  for $i := 1$ to n do
    $m[i,i] := 0$
  for $l := 2$ to n do
    for $i = 1$ to $n - l + 1$ do
      $j = i + l - 1$
      $m[i,j] := \infty$
      for $k := i$ to $j - 1$ do
        $q := m[i,k] + m[k+1,j] + d_{i-1} d_k d_j$
        if $q < m[i,j]$ then
          $m[i,j] := q$
  return m.

Longest Common Subsequence

input: sequence $X = \langle x_1, \dots, x_m \rangle$ $Y = \langle x_1, \dots, x_n \rangle$

notation: $X_i = \langle x_1, \dots, x_i \rangle$

$c[i,j]$ length of LCS of X_i and Y_j

Algorithm LCS(X,Y)

```
new ~~array~~ table C[0...m, 0....n]
for i:=0 to m do
    C[i,0]:=0
for j:=0 to n do
    C[0,j]:=0
for i:=1 to m do
    for j:= to n do
        if x_i = y_j then
            C[i,j] = C[i-1, j-1] +1
        else
            C[i,j]:= max(C[i,j-1], C[i-1,j])
return C.
```

Knapsack.

Problem: Given: a set S with n items, every item has weight w_i and benefit b_i. Maximum total weight W

Goal: take items $T \subseteq S$ such that $\sum_{i \in T} b_i$ maximal, and $\sum_{i \in T} w_i < W$

Notation: S_k contains elements 1 to k.

~~B[k,w]: is best selection inbought from S_k with weight w.~~

~~Goal: find B[n, W]~~

~~B[k,w]: the highest possible benefit from selection S_k, when you've got w "room left"~~

~~If $w_k > w$~~

~~Suppose you have B[k-1, w], what is B[k, w]?~~

$B[k, w]$ = highest possible benefit from S_k, when max weight is w.

Suppose:
$B[k-1, w]$ is known, what is $B[k, w]$?

if $k > w$, than item doesn't fit

if $k \leq w$, than $B[k,w] = \max\{B[k-1,w], B[k-1, w-w_k] + b_k$

time complexity in nW

Pseudo-polynomial algorithm:

(polynomial algorithm not expected to be found)

```
Algorithm KnapSack 1 (S, W)
  new B[0...n, 0... W]
    for w:=0 to W do
      B[0,w]:=0
    for k:=1 to n do
      B[k,0]:=0
      for w:=1 to W do
        if w_k ≤ w
          B[k,w]=max(B[k-1,w], B[k-1,w-w_k]+b_k)
        else
          B[k,w]=B[k-1,w]
```

Alternative where only one array is used

```
Algorithm knapsack 2 (S, W)
  for w:=0 to W do
      B[w]:=0
  for k:=1 to n do
      for w:= W, ..., w_k do
```

```
        if B[w-wk] + bk > B[w] then
                B[w] := B[w-wk] + bk.
return B[W]
```

Fractional Knapsack.

Problem: given a set S with n items, every item weight w_i and benefit b_i, maximum total weight W.
Goal: take fractions x_i of all items such that $\sum_{i \in S} b_i \cdot \frac{x_i}{w_i}$ is maximal, and $\sum_{i \in S} x_i \leq W$

Algorithm
Idea: take in each step as much as possible from the item i with b_i/w_i maximal. (That is: take greedy choice in each step)

```
Algorithm fractionalKnapsack (S, W)
  for each item i ∈ S do
      x_i := 0
      v_i := b_i / w_i
  w := 0
    while w < W do
    remove from S an item i with heighest value index
    a := min {w_i , W - w}
    x_i := a
    w := w + a.
```

Correctness.
Suppose A is a solution. Assume A is not greedy. That means there are i and k such that $\frac{b_i}{w_i} > \frac{b_k}{w_k}$ (i is better), and $x_i < w_i$ and $x_k > 0$ (A takes some of k) let $a = \min\{x_k, w_i - x_i\}$. Take $x_k' = x_k - a$ and

$\cancel{x_i' = x_i + a}$

$\cancel{\frac{b_i x_i}{w_i} + \frac{b_k x_k}{w_k} \leq \frac{b_i (x_i + a)}{w_i} + b_k (x_k}$ Bewijs op slides is raar.

A is not optimal $\lightning$

time complexity: analysis through priority queue and heap gives us: $O(n \log n)$

Activity Selection

Given: Set S of activities a_i each with start time s_i and finish time f_i, $s_i < f_i$ $\forall i$.
Definition compatible: two activities i,k are compatible if $f_i \leq s_k$ or $f_k \leq s_i$
Goal: Find maximum-size set of mutually compatible activities

Algorithm

~~Problem~~

S_{ij}: set of activities that start after a_i and finish before a_j. Wanted: A_{ij} maximum size sub-set of S_{ij} of mutually compatible activities.
Suppose A_k is in A_{ij}, then (informally) ~~A_{ij}~~
$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$. Dynamic Programming: take best of this, by checking all possible a_k.
Optimal substructure: A_{ik} and A_{kj} are solutions of subproblems S_{ik} and S_{kj}.

Correctness of greedy algorithm:
let S be an activity-selection problem and let a_1 be an activity with smallest finish time.
We show: $\exists$ solution containing a_1.
let A be a solution.
- if $a_1 \in A$, done
- if $a_1 \notin A$, then consider $a_k \in A$ with smallest finish time. Because $f_1 \leq f_k$, the set $A \setminus \{a_k\} \cup \{a_1\}$ is also a solution.

A9

// s: starting times
// f: finishing times monotonically increasing in finish times.

Algorithm activitySelector (s, f)
    n := s.length
    A := {a₁}
    k := 1
    for m = 2 to n do
        if s[m] ≥ f[k] then   // take first with starting time ≥ finishing time last one
            A := A ∪ {a_m}
            k := m

Huffman codes

We want to encode set of characters in an optimal way. Every character has certain frequency. Optimal: heighest frequency, ~~lowest amount~~ shortest code. ~~We want code without~~ We want prefix code: where no codeword is prefix of another.

Represent as binary tree, left is 0, right is 1, full binary tree is optimal.

Algorithm HuffmanCode(C)
    n := |C|
    Q := C
    for i := 1 to n-1 do
        new node z:
        z.left := x := removeMin(Q)
        z.right := y := removeMin(Q)
        z.freq := x.freq + y.freq
        insert (Q, z)
    return removeMin(Q)

Worst time complexity: $O(n \log n)$.

Graph Traversals.

Starting at distinguished ~~no~~node called root.
A node is first discovered and later visited.
2 methods: breadth-first and depth-first
Both in $\Theta(m)$ for m edges

Algorithm breadthFirst. discovered nodes go in queue

```
q = {root}                    // q is queue of nodes
black[w] = false              // w is node, true means visited.
while q ≠ ∅ do
    v = dequeue(q)
    black[v] = true
    for all edges vw do
        if black[w] = false then
            enqueue(w,q)
```

Algorithm depthFirst discovered nodes go in stack.

```
s = {root}
black[w] = false
while s ≠ ∅ do
    v = pop(s)
    black[v] = true
    for all edges vw do
        if black[w] = false then
            push(w,s)
```

Strongly connected component

Given: directed graph $G=(V,E)$

Def: A strongly connected component of G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v from C there

is a path from v to u.
Finding this can be done as application of depth-first search
~~Problem: Efficient~~
Use depth-first search annoted: with ~~search~~ time-stamps
- d for discovery time
- f for finish time.

~~Dijkstra's Algorithm~~
<u>Single-source shortest path</u>
$G = (V, E)$ (weighted and directed), and source node.

<u>def</u>: path from v_0 to v_n: list of vertices $p = \langle v_0, \dots, v_n \rangle$
<u>def</u>: weight of p: $\sum_{i=1}^{n} w(v_{i-1}, v_i)$
<u>def</u>: shortest-path weight $\delta(u,v) = \min\{w(p) \mid \text{path from } u \text{ to } v\}$, if there is a path from u to v. Otherwise $\delta(u,v) = \infty$.
<u>def</u>: shortest path from u to v: path with weight $\delta(u,v)$

<u>~~Sing~~ Problem: give for every node v a shortest path from the source to v.</u>

- <u>Initialization</u>: set all distances to ∞ except for source.

Algorithm initialize (G, S)
  for every $v \in V$
    $v.d := \infty$
    $v.\pi := nil$
  $S.d := 0$

Initialization is in $\Theta(|V|)$

- relaxation

Test whether we can improve the shortest path so far.

<u>Algorithm</u> relax (u, v, w)

```
if v.d > u.d + w(u,v) then
    v.d = u.d + w(u,v)
    v.π = u.
```

Relaxation is in $O(1)$

- Bellman Ford Algorithm. (returns true if no negative weight cycle)

```
Algorithm BellmanFord (G, w, s)
  initialize (G, s)
  for i = 1 to |G.V| - 1 do
      for each (u,v) ∈ G.E do
          relax (u,v,w)
  for each (u,v) ∈ G.E do
      if v.d > u.d + w(u,v) then
          return false
  return true.
```

When there are no negative weights you can also use Dijkstra:

```
Algorithm Dijkstra (G, w, s)
  initialize (G, s)
  S := ∅
  Q := G.V
  while Q ≠ ∅ do
      u := extractMin (Q)
      S: S ∪ {u}
      for each v ∈ G.adj [u] do
          relax (u,v,w)
```

All

Properties Dijkstra:
- greedy
- insertion, decrease key and extraction from priority queue
- V times an extractMin operation : $O(V \log V)$
- E times a decreaseKey operation: $O(E \log V)$

$O((V+E)\log V) = O(V \log V)$ if all nodes reachable.

String Matching Problem.

def: String: sequence of characters $P[1, \dots, m]$
def: prefix: initial part : $P[1, \dots, k]$ $0 \leq k \leq m$
def suffix: final part : $P[k, \dots, m]$ $0 \leq k \leq m$
def substring: subarray.

Problem: Given text $T[1, \dots, n]$ and Pattern $P[1 \dots m]$
Question: is P in T?
If P is in T, notation is as follows:
P occurs at position s in T.
so $T[s+1, \dots, s+m] = P[1, \dots, m]$

```
Algorithm (T,P)              // Brute force.
  n := T.length
  m := P.length
  for s:=0 to n-m do
     j := 1
     while j ≤ m and T[s+j] = P[j] do
        j := j+1
     if j=m+1 then
        print 'occurs with shift s'
```

if m roughly half of n. then in $O(n^2)$

Knuth-Morris-Pratt algorithm

Idea: compare P with T from left to the right. if mismatch, ~~go~~ you don't go back to beginning but go on ~~on, remembering~~. While reading, you also keep up with new matches.

```
Algorithm KMPMatch(T,P)
  i:=1
  j:=1
  while i ≤ n do
     if P[j]=T[i] then
        if j=m then match
        i:=i+1
        j:=i+1
     else
        if j >1 then
           j:= f(j-1)+1
        else
           i:=i+1
  return fail
```

$f(\cdot)$: failure/prefix function.
For P = ababaca:

i	1	2	3	4	5	6	7
p(i)	a	b	a	b	a	c	a
f(i)	0	0	1	2	3	0	1

Time complexity:
- computing ~~fai~~ $f(\cdot)$: $O(m)$
- iteration $O(n+m)$