Info 1 Zusammenfassung

Allgemeines

Shift Operation

- x >> 1 → x/2
- x << 2 → x*4

logisch: links Nullen (bei unsigned)

arithmetisch: links Vorzeichen (bei signed)

- & Bitwise AND
- | Bitwise OR
- ^ Bitwise XOR

explicit type cast:

```
double d = 0,123;
int i = (int) d;
```

```
MLABEL: if...
        ....
goto MLABEL;
```

Gotos

```
if ... if ... else ...
```

"dangling else problem"

```
switch (variable)
{
  case 1: ... break;
  default: ... break;
}
```

Switch-case

a=b

Ausdruck mit Wert a

Static storage → globale vars

Stack → lokale vars

```
do {
...
} while (...);
```

do-while

Heap: (5000+2)

```
int *a = (int*) malloc(sizeof(int));
```

save address into pointer · cast *void to *int · how many bytes

```
free(a);
```

Array indices

v[i] = *(v+i) = *(i+v) = i[v]

p=v ≙ p=&v[0] → address of first element

*p++=i; ≙ *p=i; p++;

pointer+increment

	Mo	Di	...
1	...	...	...
2	...	...	...
...	...	...	...

⇒ 2D Array

```
enum tag {Mo, Di, ...}
```

→ "Mo = 0, Di = 1, ..."

Structs

```
struct point {
    float x, y;
};

struct point p1;

p1.x = 0;
p1.y = 0;
```

```
typedef struct Point {
    float x, y;
} Point;

Point p1;

p1.x = 0;
p1.y = 0;
```

p1=p2; kopiert alle Komponenten

p.x = ... ; // direkt

p_ptr -> x = ... // über pointer

(*p_ptr).x = ...

Info 1 Zusammenfassung Fortsetzung 1 | Laufzeit

kartesisches Produkt der Mengen A, B

$\boxed{A \times B = \{(a,b) \mid a \in A, b \in B\}}$

Relation R

$\boxed{R \subseteq A \times B}$

Funktion $\boxed{F: A \to B}$ Relation, max. 1 b für jedes a

```
...
int f();
a = f();
int f(){return 1}
...
```

Funktionen: erst deklarieren dann verwenden (definieren geht auch später)

Funktion als Parameter:

```
int op(int (*f)(int,int), ...){...}
```

- keine Arrays können zurückgegeben werden
- structs by default call by value (ineffizient)

O-Notation

$f: \mathbb{N} \to \mathbb{N}, \ g: \mathbb{N} \to \mathbb{N}$

$\boxed{f \in O(g)}$ falls $\boxed{\exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R} \ \forall n > n_0 : f(n) \leq c \cdot g(n)}$

"f wächst nicht schneller als g ab n_0"

$f + g \in O(\max\{f, g\})$
$f \cdot g \in O(f \cdot g)$

fast pow
(n gerade → $(b^{n/2})^2$
n ungerade → $b \cdot b^{n-1}$)
→ $\boxed{O(\log n)}$

lineare Suche $\boxed{O(n)}$

(vorsortiert) binäre Suche $\boxed{O(\log n)}$

divide and conquer

```
int binsearch(int A[]
     int key, int l, int r){
  int k;
  while (r>=l){
    k = (l+r)/2;
    if(key==A[k]) return k;
    if(key < A[k]) r =k-1;
    else l=k+1;
  }
  return -1;
}
```

Info 1 Zusammenfassung Fortsetzung 2

Sortieren

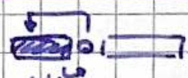
```
void selection_sort(int A[], int l, int r){
    int i, j, min;
    for(i=l; i<r; i++){
        min=i;
        for(j=i+1; j<=r; j++){
            if (A[j]<A[min]) min=j;
        }
        exchange(A[i], A[min]);
    }
}
```

"kleinstes aussuchen und vorne einfügen"

B.C., a.C., w.C. } $\boxed{O(n^2)}$

stabil? nein

```
void insertion_sort(int A[], int l, int r){
    int i, j, v;
    for(i=l+1; i<=r; i++){
        v=A[i];
        for(j=i-1; j>=l; j--){
            if(v<A[j]) A[j+1]=A[j];
            else break;
        }
        A[j+1]=v;
    }
}
```



"an die richtige Stelle einfügen, die anderen verrücken"

B.C. $\boxed{O(n)}$

a.C., w.C. } $\boxed{O(n^2)}$

stabil? Ja

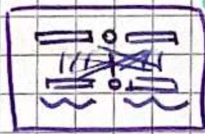
```
void bubble_sort(int A[], int l, int r){
    int i,j;
    for(i=l; i<r; i++){
        for(j=r; j>i; j--){
            if (A[j-1]>A[j]){
                exchange(A[j-1], A[j]);
            }
        }
    }
}
```

B.C. $\boxed{O(n)}$

a.C., w.C. } $\boxed{O(n^2)}$

stabil? Ja

Info 1 Zusammenfassung Fortsetzung 3 Sortieren



```
void quick_sort(int A[], int l, int r){
    int k;
    if(r<=l) return;
    k = partition(A, l, r);
    quick_sort(A, l, k-1);
    quick_sort(A, k+1, r);
}

int partition(int A[], int l, int r){
    int i, j, k, v;
    k=r;              } pivot = letztes
    v = A[k];
    i=l;
    j=r-1;
    while(true){
        while(i<r && A[i]<=v) i++;    } von beiden Seiten laufen,
        while(j>=l && A[j]>=v) j--;   } "falsche" Elemente suchen
        if(i>=j) break;
        else exchange(A[i], A[j]);    } "falsche" vertauschen
    }
    exchange(A[i], A[k]);    } pivot in die Mitte setzen
    return i;
}
```

b.C., a.C.: $\boxed{\mathcal{O}(n \cdot \log n)}$

w.C.: $\boxed{\mathcal{O}(n^2)}$

stabil? Nein

Info 1 Zusammenfassung Fortsetzung 4 | Sortieren

```
void heap_sort(int A[], int N){
    int i;
    for (i=N/2; i>0; i--){      } Heap bauen
        sink(A, i, N);           } bis N/2, ab da
    }                            } keine Kinder
    for (i=N; i>1; i--){
        exchange(A[1], A[i]);   } Min ans Ende
        sink (A, 1, i-1);       } Heap reparieren
    }                 nicht N,
}                     ab da ist schon sortiert
```

A[0] unbenutzt

b.c.	
a.c.	$\boxed{\mathcal{O}(n \cdot \log n)}$
w.c.	

stabil? Nein

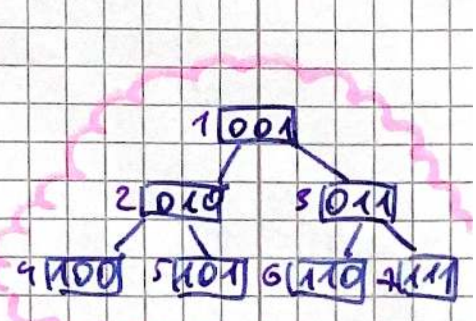
```
void sink (int A[], int k, int N){
    int child;
    while (true){
        if(2*k > N) break;        } kein Kind
        if(2·k+1 <= N){
            if (A[2·k] < A[2·k+1]) child = 2·k;   } zwei Kinder
            else child = 2·k+1;                   } → nehme das kleinere
        } else child = 2·k;       } ein Kind links

        if (A[k] > A[child]){
            exchange(A[k], A[child]);   } geht runter
            k = child;                  } mache unten weiter
        } else break;                   } genug eingesunken
    }
}
```

(Minimums-) Heap:

$$\forall A[i] : \quad A[i] \leq A[2i]$$
$$A[i] \leq A[2i+1]$$

$\underbrace{A[i]}_{\text{Element}} \leq \underbrace{A[2i+1]}_{\text{Kinder}}$ → Darstellung des Array als Baum



Info 1 Zusammenfassung Fortsetzung 5 — Linked Lists

"Verkettete Liste": **Linked List**: leer oder Referenz auf Knoten, der ein Element sowie eine Referenz auf eine Linked List enthält

```
typedef struct list {
    nodeptr first, last;
} List, *listptr;
```

```
typedef struct node {
    T* data;
    struct node *next;
} Node, *nodeptr;
```

```
void Init(listptr L);
```

```
int IsEmpty(List L);
```

```
nodeptr newnode(T* item){
    nodeptr np;
    np=(nodeptr)malloc(sizeof(Node));
    np->data =item;
    np->next =NULL; return np;}
```

```
void AppendFirst(T* item, listptr L){
    nodeptr np= newnode(item);
    if(IsEmpty(*L)){
        L->first =np; L->last =np;      } Empty
    }else{
        np->next = L->first;            } not Empty
        L->first =np;
    }
}
```

196

b.c. $\boxed{\Theta(1)}$
a.c., w.c. $\boxed{\Theta(n)}$

```
void AppendLast(T* item, listptr L){
    nodeptr np=newnode(item);
    if (IsEmpty(*L)){
        L->first =np; L->last = np;     } Empty
    }else{
        L->last->next =np;              } not Empty
        L->last =np;
    }
}
```

```
int IsIn(T* item, List L){
    nodeptr np;
    if(IsEmpty(L)) return 0;
    np = L.first;
    while (np!=NULL){
        if (Equal(np->data, item)){
            return 1;
        }
        np = np->next;
    }
    return 0;
}
```

```
listptr Union(listptr L1, listptr L2){
    if(IsEmpty(*L2)) return L1;
    if(IsEmpty(*L1)) return L2;
    L1->last->next =L2->first;
    L1->last = L2->last;
    return L1;
}
```

Info 1 Zusammenfassung Fortsetzung 6

Linked Lists

Linked Lists: Fortsetzung

```
// 2 hinter ^
void InsertBehind(T* item1, T* item2, listptr L){
    nodeptr np, newnp;
    if (!IsIn(item1, *L)){
        printf("Fehler!"); return;      } item 1 not even in list
    }
    np = L->first;
    while (np != NULL){
        if(Equal(np->data, item1)){
            newnp = newnode(item2);
            newnp->next = np->next;
            np->next = newnp;
            if(np == L->last){
                L->last = newnp;        } item 1 is last
            }
            break;
        }
        np=np->next
    }
}
```

```
void Delete(T* item, listptr L){
    nodeptr np1, np2;
    if (IsEmpty(*L)) return;     } L Empty
    np1 = L->first;
    if (Equal(np1->data, item)){
        L->first = np1->next;
        if (L->first == NULL){
            L->last = NULL            } item is first
        }
        free(np1); }!
        return;
    }
    np2 = np1->next;              np2 is deleted,
    while(np2 ≠ NULL){            np1 (previous) stays
        if(Equal(np2->data, item)){
            np1->next = np2->next;
            if(np2 == L->last){
                L->last = np1;
            }
            free(np2); }!
            break;
        }
        np1 = np2;
        np2 = np2->next;
    }
}
```

```
listptr Intersect(
    listptr L1, listptr L2){
    nodeptr np;
    listptr L = (listptr) malloc(
                    sizeof(list));
    Init(L);
    np = L1->first;
    while (np ≠ NULL){
        if(IsIn(np->data,
            *L2)){
            AppendLast(np->data,
                L);
        }   np = np->next;
    }
    return L;
}
```

Info 1 Zusammenfassung Fortsetzung 7

Stack

Stack: ADT, die 'Push' und 'Pull' kann (LIFO)

Stack durch Liste

```
List L;
void Push(T* item, listptr L){
    AppendFirst(item, L);
}

T* Pop(listptr L){
    T* item;
    if (IsEmpty(*L)) return NULL;
    item = L->first->data;
    Delete(item, L);
    return item;
}
```

Stack durch Array

```
#define MAX 100
typedef struct stk{
    T* elems[MAX]; int top;
}stack;

void Init(stack *s){
    s->top=0;
}

int IsEmpty(stack s){
    return (s.top ==0);
}

void Push(T* item, stack *s){
    if (s->top == MAX) {printf("Stack full!"); return;}
    s->elems[s->top]=item;
    s->top++;
}

T* Pop(stack *s){
    if (IsEmpty(*s)) return NULL;
    s->top--;
    return s->elems[s->top];
}
```

Info 1 Zusammenfassung Fortsetzung 8 Queue

Queue: ADT, die 'Put' und 'Get' kann (FIFO)

Queue durch Liste

```
List L;
void Put(T* item, listptr L){
    AppendLast(item, L);
}
T* Get(listptr L){
    T* item;
    if(IsEmpty(*L)) return NULL;
    item = L->first->data;
    Delete(item, L);
    return item;
}
```

Queue durch Array

```
#define MAX 100
typedef struct q{
    T* elems[MAX];
    int first, last;
} queue;
void Init(queue *q){
    q->first = 0;
    q->last = 1;
}
void Put(T* item, queue *q){
    if(q->last == q->first) overflow();
    else {
        q->elems[q->last] = item;
        q->last = (q->last+1) % MAX;
    }
}
T* Get(queue *q){
    q->first = (q->first + 1) % MAX;
    if (q->first == q->last) underflow();
    else return q->elems[q->first];
}
```

Info 1 Zusammenfassung Fortsetzung 9

Baum

Baum: Knoten r und $k \geq 0$ disjunkte Bäume

Binärbaum: $B = (r, B_L, B_R)$: leer oder Wurzel + Links/rechts

```c
typedef struct tr{
    T* data;
    struct tr *left, *right;
} btree, *btreeptr;
```

Durchlauf:
- Inorder LWR
- Preorder WLR
- Postorder LRW

```c
void inorder(btreeptr b){
    if (b==NULL) return;
    inorder(b->left);
    visit(b->data);
    inorder(b->right);
}
```

...

ausgeglichen
suche $O(\log N)$

entartet: Liste

AVL-Baum: ausgeglichen binär Suche

Mengen → durch Liste / durch Baum → suchen: in $O(N)$ / ac. $O(\log N)$ → Binäre Suche

Suchbaum: Binärbaum, wo links < Wurzel < rechts

```c
btreeptr insert(T* item, btreeptr b){
    btreeptr n;
    if(b == NULL){
        n = (btreeptr) malloc(sizeof(btree));
        n->data = item;
        n->left = NULL; n->right = NULL;
        return n;
    }                                        // Blatt
    if (key(item) < key(b->data)){
        b->left = insert(item, b->left);     // links
    }
    if(key(item) > key(b->data)){
        b->right = insert(item, b->right);   // rechts
    }
    return b; } schon drin oder schon eingefügt
}
```

```c
T* search(T* item, btreeptr b){
    if (b == NULL) return NULL;
    if(key(item) == key(b->data))
        return b->data;
    if(key(item) > key(b->data))
        return search(item, b->right);
    ...<
}
```

Info 1 Zusammenfassung Fortsetzung 10

Graph

Relation → $R \subseteq A \times B$

(gerichteter) Graph → $G = (V, E)$:
- Menge V von Knoten (vertices)
- Menge E von Kanten (edges) $E \subseteq V \times V$

ungerichteter Graph: für alle $(u, v) \in E \Rightarrow (v, u) \in E$

Vorgänger $\{u, v\} \in E$ Nachfolger

Pfad: Folge von Knoten $(v_1, v_2, \dots, v_n)$, für alle $i = 1, \dots, n-1$
$\{v_i, v_{i+1}\} \in E$

durch eine Kante
verbundene Knoten: adjazente Knoten

zusammenhängender Graph: zw. bel. $u, v \in V$ existiert mind. ein Pfad

zyklenfreier Graph: zw. bel. $u, v \in V$ existiert höchstens ein Pfad

Baum: zusammenhängender zyklenfreier Graph

(kanten) gewichteter Graph $G = (V, E, w)$ mit $w: E \to \mathbb{R}$

Dijkstra

$O(|V|^2)$

fertig — bekannt, nicht bearbeitet — kurze Kanten

```
void shortest_path(Graph G=(V,E)){
  set BLUE={}, GREEN={v0}, ORANGE={};
  dist[v0]=0;
  while(GREEN != {}){
    v=MinDist(GREEN);          <- such den am wenigsten Entfernten
    Insert(v, BLUE);           <- er ist jetzt bearbeitet
    Delete(v, GREEN);
    for (u ∈ Succ(v)){         <- gehe seine Nachfolger durch
      if(!(u ∈ GREEN ∪ BLUE)){         | unbekannt
        Insert({u,v}, ORANGE);         | -> erstmal in GREEN rein
        Insert(u, GREEN);
        dist[u]=dist[v]+cost(u,v);
      } else if (u ∈ GREEN){                           ↓ bekannt -> bearbeite
        if(dist[v]+cost(u,v) < dist[u]){               ↓ Pfad über v kürzer?
          Insert({u,v}, ORANGE);                       -> Kante gut
          e=previousEdge(u);                           | -> alte Kante schlecht
          Delete(e, ORANGE);
          dist[u]=dist[v]+cost(u,v);                   -> neue Entfernung
        }
      }
    }
  }
}
```

dist[v] = aktueller min. Abstand
cost(u,v) = Gew. d. Kante

Info 1 Zusammenfassung Fortsetzung 11

Graph

Adjazenzmatrix zu $G=(V,E)$:

$(n \times n)$-Matrix $A = a_{ij}$ mit $a_{ij}=1$ für $\{v_i, v_j\} \in E$ (Kante)
$a_{ij}=0$ sonst (keine Kante)

- Prüfung der Adjazenz in $\boxed{O(1)}$
- Platzbedarf $\boxed{O(|V|^2)}$

Adjazenzliste: Linked List aller Nachbarn für jeden Knoten

- Prüfung der Adjazenz w.c. $\boxed{O(|V|)}$
- Platzbedarf $\boxed{O(|V|+|E|)}$

DFS: Tiefensuche

Adjazenzlisten: $\boxed{O(|V|+|E|)}$

```
int visited[N];
void dfs(int v){
    int w;
    visited[v] = 1;   |setze v als besucht
    process(v);
    for( {v,w} ∈ E ){   ↓ für jeden Nachbarn von v
        if(!visited[w]) dfs(w);   |geh tiefer falls noch nicht besucht
    }
}
```

meistens erst die kleinen

BFS: Breitensuche

Adjazenzlisten: $\boxed{O(|V|+|E|)}$

```
int visited[N];
void bfs(int v){
    int w; Queue Q;     | v besucht
    visited[v] = 1;     |   init Queue
    process(v);
    Put(v,Q);
    while (!IsEmpty(Q)){
        v = Get(Q);               |pull new el from Queue
        for ({v,w} ∈ E){          ↓ für jeden Nachbar
            if (!visited[w]){     der noch nicht besucht wurde
                visited[w] = 1;   
                process(w);       } besuche und tue ihn
                Put(w,Q);           in die Queue
            }
        }
    }
}
```

Info1 Zusammenfassung Fortsetzung 12

Graph

Spannbaum: im ungerichteten zusammenhängenden kantengewichteten Graphen $G=(V,E,w)$ zyklenfreier zusammenhängender Teilgraph $\boxed{G'=(V',E',w) \text{ mit } V=V' \text{ u. } E' \subseteq E}$

minimaler Spannbaum: Spannbaum mit minimaler Kantengewichtssumme unter allen Spannbäumen

Kruskal-Algorithmus

Greedy

vom sortieren, while-Schleife linear

$\boxed{\Theta(|E| \cdot \log|E|)}$

```
Graph MinSpanningTree(Graph G=(V,E)){
    Graph T; List L; Edge e;
    T={};
    L=Sort(E);  | aufsteigende Liste der Kanten
    while (#Kanten in T)< #Knoten in G)-1 && !IsEmpty(L)){
        e=FirstElem(L);
        Delete(e,L);
        if(!Cycle(e,T)){   | kleinste Kante hinzufügen,
            T=T ∪ {e};     | falls das keinen Zyklus
        }                  | verursachen würde
    }
    if (#Kanten in T)< #Knoten in G)-1) {
        printf("Nicht zusammenhängend");
    }
    return T;
}
```

wir brauchen #V-1 Kanten

Infor Zusammenfassung Fortsetzung 13

Techniken

Rekursion

- ⊕ elegant u. kompakt
- ⊖ höherer Speicherbedarf
- ⊖ Gefahr von infinite loops

Iteration

! Rekursion ↯

⊕ Schleifen können durch Compiler optimiert werden

Backtracking

$\boxed{\text{Teillösung} \rightarrow \text{Gesamtlösung}}$ Analogie: DFS

- Raten falls Weg zur Lösung unbekannt
- frühere Entscheidungen zurücknehmen, falls sie nicht funktionieren
- notfalls alle Möglichkeiten ausprobiert
- z.B. Testen von Schaltungen

Kombinatorische Optimierung

- Optimale Lösung aus einer Menge der Lsgs aussuchen
- z.B. minimaler Spannbaum

→ Greedy: beste Teillösung zum aktuellen Zeitpunkt

⊕ geringe Laufzeit ⊖ nicht immer optimale Lösung

z.B. Kruskal

Heuristiken

N-P-vollständig: kombinatorische Optimierungsprobleme, die nicht in akzeptabler Zeit gelöst werden können

- z.B. Travelling Salesman Problem
- gibt es x für f(x), sodass f(x)=1

→ Approximationsalgorithmen (Effi)

Dynamische Programmierung: z.B. Befehlsauswahl

Optimierungsprobleme

Zerlegung des Problems in Teilprobleme, Lösung dieser, Speicherung der Teillösungen, um erneute Berechnung zu vermeiden.

Branch & Bound:

Entscheidungsbaum, aber Wege, die offensichtlich nicht zum Ergebnis führen, werden direkt verworfen.

→ $\boxed{\text{Schranken}}$ berechnen (lower & upper bound)

→ muss schnell u. eindeutig gehen