Aufgabe 1

main.cpp: cpp

```cpp
#include "sort.cpp"
#include <iostream>

int main() {
    int a[8] = {3,2,2,34,12,5,0,9};
    int b[8] = {3,2,2,34,12,5,0,9};

    for (int i = 0; i < 8; i++)
        std::cout << a[i] << " ";
    std::cout << std::endl;

    countSort(a, 8, 34);

    for (int i = 0; i < 8; i++)
        std::cout << a[i] << " ";
    std::cout << std::endl;

    for (int i = 0; i < 8; i++)
        std::cout << b[i] << " ";
    std::cout << std::endl;

    countSortFaster(b, 8, 34);

    for (int i = 0; i < 8; i++)
        std::cout << b[i] << " ";
    std::cout << std::endl;
}
```

linked_list.hpp:

```cpp
#pragma once
#include <iostream>

//Note that this countsort only works on positive Values (no negatives allowed because no negative index is possible)
void countSort(int* a, int n, int k) {
    int count[k+1] = { 0 };

    for (int i = 0; i < n; i++)     //Theta(n)
        count[a[i]]++;

    int currentIndex = 0;
    for (int i = 0; i <= k; i++) {  //Theta(k)
        while (count[i] != 0) {    //Theta(3n)
            a[currentIndex] = i;
```

```
            currentIndex++;
            count[i]--;
        }

    }
}

void countSortFaster(int* a, int n, int k) { //Theta(2k + 3n)
    int count[k+1] = { 0 };                  //Theta(k+1) = Theta(k)
    int res[n];

    for (int i = 0; i < n; i++)              //Theta(n)
        count[a[i]]++;

    int idx = 0;
    for (int val = 0; val <= k; val++) {     // Theta(k)
        while(count[val]--)                  // Theta(n)
            a[idx++] = val;                  // Theta(n)
    }
}
```

Aufgabe 2

main.cpp: cpp

```cpp
#include "linked_list.cpp"
#include <iostream>

int main() {
    linked_list<int> a;

    int b[10] = {2,1,3,45,5,-5,-1,10,-10,6};

    for (int i = 0; i < 10; i++) {
        a.enqueue(b[i]);
    }

    a.print();
    a.Quicksort();
    a.print();
}
```

linked_list.hpp:

```
#include <stdexcept>
#include <iostream>
template<typename listT> class linked_list {
    /*
```

```
     * Keep in mind new node uses CopyConstructor and assignement operator
     */
    template<typename nodeT> struct node {
        node* next;
        nodeT val;

        node(nodeT value): next(nullptr) {
            val = nodeT(value);
        }

        friend std::ostream& operator<< (std::ostream& out, const node& in) {
            return out << in.val;
        }
    };

    using element=node<listT>;

    element* m_head;
    element* m_tail;
    unsigned int len = 0;

    void QuickSortRec(int f,int l) {
        auto swap = [] (int &a, int &b) {
            int tmp = a;
            a = b;
            b = tmp;
        };
        auto PreparePartition = [swap] (linked_list<listT>& a, int f, int l, int &p)
        {
            int pivot = a[f];
            p = f-1;

            for (int i = f; i <= l; i++)
            {
                if (a[i] <= pivot){
                    p++;
                    swap(a[i],a[p]);
                }
            }
            swap(a[f],a[p]);
        };

        int part;
        if (f<l) {
            PreparePartition(*this,f,l,part);
            QuickSortRec(f,part-1);
```

```
            QuickSortRec(part+1,l);
        }
    };
public:

    void operator=(const linked_list<listT>&) = delete;

    linked_list(): m_head(nullptr), m_tail(nullptr) {}
    ~linked_list() {
        this->flush();
    }
    const listT& Head () const {
        if (m_head != nullptr)
            return m_head->val;
        throw std::runtime_error("cannot return head of empty list");
    }
    void enqueue(listT& newIn) {
        if (m_head == nullptr) {
            m_head = new element(newIn);
            m_tail = m_head;
        } else {
            m_tail->next = new element(newIn);
            m_tail = m_tail->next;
        }
        len++;
    }
    listT dequeue() {
        if (m_head != nullptr) {
            element* tmp = m_head->next;
            listT res = m_head->val;
            delete m_head;
            m_head = tmp;
            len--;
            return res;
        }
        throw std::runtime_error("cannot dequeue empty list");
    }
    void flush() {
        while (m_head != nullptr) {
            element *tmp = m_head->next;
            delete m_head;
            m_head = tmp;
        }
        m_tail = nullptr;
        len = 0;
    }
```

```
    void print() const {
        for (element *tmp = m_head; tmp != nullptr; tmp = tmp->next) {
            std::cout << *tmp << " ";
        }
        std::cout << std::endl;
    }

    listT& operator[] (int index) {
        if (index >= len)
            throw std::runtime_error("index out of bounds");
        element *res = m_head;
        for (int i = 1; i <= index; i++)
            res = res->next;
        return res->val;

    }

    void Quicksort() {
        QuickSortRec(0, len-1);
    }
};
```

- Speicher (im Vergleich mit einem normalen C-Style Arrays):

1. Verkette Liste:
   - Speicher:
     - Speicher pro Node: $\mathrm{S}_{\mathrm{Node}}(\mathrm{NodeT}) = 8\ \mathrm{Byte(Pointer)} + \mathrm{sizeof(NodeT)}$
     - Speicher insgesamt bei n Elementen:

       $\mathrm{S}_{\mathrm{ges}}(\mathrm{NodeT}) = n\mathrm{S}_{\mathrm{Node}}(\mathrm{NodeT}) + 2 \cdot 8\ \mathrm{Byte(Head\ und\ Tail\ pointer)} + 4\ \mathrm{Byte(len)}$

       $\Rightarrow \mathrm{S}_{\mathrm{ges}} \in \Theta(n)$
   - Laufzeit:
     - Quicksort:

       $\mathrm{T}^{\mathrm{BC}}_{\mathrm{QuickSort}} = n \log n$

       $\mathrm{T}^{\mathrm{AC}}_{\mathrm{QuickSort}} = n \log n$

       $\mathrm{T}^{\mathrm{WC}}_{\mathrm{QuickSort}} = n^2$
     - Verkette Liste: Indexierung über Operator[] an i-ter Stelle

       $\mathrm{T}_{[]}(i) = i$

       $\Rightarrow$ Pro PreparePartition addiert sich ein Faktor $\lambda : \sum_{i=f}^{l} i + 1 \leq \lambda \leq 2\sum_{i=f}^{l} i + 1$

       $\Rightarrow T(n) = 2T(\frac{n}{2}) + n + \lambda$

2. C-Style Arrays:
   - Speicher: $\mathrm{len(Array)} = n \cdot \mathrm{sizeof(type)} \Rightarrow \mathrm{S}_{\mathrm{ges}} \in \Theta(n)$
   - Laufzeit: wie bei normalen QuickSort

     $\mathrm{T}^{\mathrm{BC}}_{\mathrm{QuickSort}} = n \log n$

$$\mathrm{T}^{\mathrm{AC}}_{\mathrm{QuickSort}} = n \log n$$
$$\mathrm{T}^{\mathrm{WC}}_{\mathrm{QuickSort}} = n^2$$

Aufgabe 3

main.cpp: cpp

```cpp
#include "ring_list.cpp"
#include <cstdlib>
#include <iostream>
#include <random>

                                                //Speicherplatz: Theta(n)
int main() {                                    //Laufzeit: Theta(n^2 + 6 +
index + unknown) wobei index element of [1,49] => Theta(n^2 + unknown)
    ring_list values;
    for (int i = 1; i < 50; i++)                //Theta(n^2)
        values.enqueue(i);

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(1,49);

    for (int i = 0; i < 6; i++) {               //Theta(6)
        int index = distrib(gen);               //Theta(unknown)
        std::cout << values.dequeueAt(index) << " "; //Theta(index)
    }
    std::cout << std::endl;
}
```

linked_list.hpp:

```cpp
#include <stdexcept>
#include <iostream>
class ring_list {
    /*
     * Keep in mind new node uses CopyConstructor and assignement operator
     */
    struct node {
        node* m_next;
        int val;

        node(int value, node* next): m_next(next), val(value) {}
        friend std::ostream& operator<<(std::ostream& out, const node& in) {
            return out << in.val;
        }
    };


    node* m_head;
    node* m_tail;
public:
```

```
    void operator=(const ring_list) = delete;

    ring_list(): m_head(nullptr), m_tail(nullptr) {}
    ~ring_list() {
        this->flush();
    }
    const int& Head () const {
        if (m_head != nullptr)
            return m_head->val;
        throw std::runtime_error("cannot return head of empty list");
    }
    void enqueue(int& newIn) {
        if (m_head == nullptr) {
            m_head = new node(newIn, m_head);
            m_tail = m_head;
        } else {
            m_tail->m_next = new node(newIn, m_head);
            m_tail = m_tail->m_next;
        }
    }
    int dequeue() {
        if (m_head != nullptr) {
            node* tmp = m_head->m_next;
            int res = m_head->val;
            delete m_head;
            m_head = tmp;
            m_tail->m_next = m_head;
            return res;
        }
        throw std::runtime_error("cannot dequeue empty list");
    }

    int dequeueAt(unsigned int index) {
        if (m_head == nullptr)
            throw std::runtime_error("cannot dequeue empty list");
        if (index == 0){
            node* next = m_head->m_next;
            int res = m_head->val;
            delete m_head;

            if (next == nullptr) {
                m_head = nullptr;
                m_tail = nullptr;
            } else {
                m_head = next;
                m_tail->m_next = m_head;
```

```
            }

            return res;
        }
        node* tmp = m_head;
        node* prev;
        for (int i = 1; i <= index; i++) {
            prev = tmp;
            tmp = tmp->m_next;
        }
        prev->m_next = tmp->m_next;
        int res = tmp->val;
        delete tmp;
        return res;
    }

    void flush() {
        if (m_head != nullptr)
            m_tail->m_next = nullptr;
        while (m_head != nullptr) {
            node *tmp = m_head->m_next;
            delete m_head;
            m_head = tmp;
        }
        m_tail = nullptr;
    }

    void print() const {
        for (node *tmp = m_head; tmp != nullptr; tmp = tmp->m_next) {
            std::cout << *tmp << " ";
        }
        std::cout << std::endl;
    }

    int& operator[] (unsigned int index) {
        node *res = m_head;
        for (int i = 1; i <= index; i++)
            res = res->m_next;
        return res->val;

    }
};
```

Aufgabe 4

RucksackProblem

```python
calcBackpack(objects[n], maxWeight):  # Theta(n) + Theta(n log n) + Theta(n) =>
calcBackpack = Theta(n log n)
  ratios = []
  for i, o in enumerate(objects):     #Theta(n)
    ratios.push((o.val/o.weight, i))
  ratios.sort()                       #Theta(n log n)

  res = (0,0)
  last = none
  addedObj = []

  for r in ratios:                    #Theta(n)
    if (res[1] >= weight):
      break;
    res[0] += r[0].val
    res[1] += r[0].weight
    addedObj.push(objects[r[1]])

  if res[1] =< weight:
    return (res, addedObj);

  res[0] -= addedObj.last().val
  res[1] -= addedObj.last().weight

  restToMaxWeight = maxWeight - res[1]
  factor = restToMaxWeight/addedObj.last().weight
  res[0] += factor * addedObj.last().val
  res[1] += factor * addedObj.last().weight

  return (res, addedObj)
```

Teil 2:

$$M = 21, [(v = 10, w = 2), b = (v = 50, w = 20)]$$

$$\Rightarrow \text{ratio}_a = \frac{10}{2} = 5, \text{ratio}_b = \frac{50}{20} = 2.5 \Rightarrow \text{nach sortieren}$$

$$\Rightarrow [\text{ratio}_a, \text{ratio}_b] \Rightarrow \text{nach algorithmus mit} \quad a_i \in \{0, 1\} \ \text{res}_v = 10$$

Das ist nicht das Optimum. Daraus folgt, dass der Algorithmus bei $a_i \in \{0, 1\}$ sehr schlecht werden kann.