# Objektorientierte Programmierung

Leif Jacob (175213)

## Introduction

1. What is the time complexity of the following code
   Die Funktion ist in O(n), da wir durch den Vektor mit n Elementen zweimal durchgehen und $2n \in O(n)$ ist.

```
void foo(const vector<int> &vec) {
  int sum = 0;
  int product = 1;
  for (int num : vec) {
    sum += num;
  }
  for (int num : vec) {
    product *= num;
  }
  cout << sum << ", " << product;
}
```

   Die Funktion ist in $O(n^2)$, da wir für jeden Eintrag vom Vektor wieder durch den ganzen Vektor gehen.

```
void print_pairs(const vector<int> &vec) {
  for (size_t i = 0; i < vec.size(); i++) {
    for (size_t j = 0; j < vec.size(); j++) {
      cout << "(" << vec[i] << ", " << vec[j] << ") ";
    }
  }
}
```

   Wir gehen für den i-ten Eintrag vom Vektor durch i weitere Elemente. Damit ergibt sich die summe $\sum_{i=1}^{n-1} i = (n(n-1)/2)$. Dies ist in $O(n^2)$ enthalten.

```
void print_unordered_pairs(const vector<int> &vec) {
  for (size_t i = 0; i < vec.size(); i++) {
    for (size_t j = i + 1; j < vec.size(); j++) {
      cout << "(" << vec[i] << ", " << vec[j] << ") ";
    }
  }
}
```

   Für jedes Element von vecA gehen wir den Vektor vecB einmal vollständig durch. Damit ist die Komplexität $O(a \cdot b)$ mit $a = vecA.size(), b = vecB.size()$.

```
void print_unordered_pairs(const vector<int> &vecA, const vector<int> &vecB) {
  for (int numA : vecA) {
    for (int numB : vecB) {
      cout << "(" << numA << ", " << numB << ") ";
    }
  }
}
```

   Wie in dem Beispiel davor nur, dass wir noch jedes mal 10 Schritte machen und somit auf $10a \cdot b$ kommen. Dies ist aber wieder in $O(a \cdot b)$ enthalten da es sich um einen konstanten Faktor handelt.

```
void print_unordered_pairs_multiple(const vector<int> &vecA,
                                    const vector<int> &vecB) {
  for (int numA : vecA) {
    for (int numB : vecB) {
      for (int i = 0; i < 10; ++i) {
        cout << "(" << numA << ", " << numB << ") ";
      }
    }
  }
}
```

Wir gehen die Hälfte alle Einträge von vec durch. Dies ist aber wieder ein konstanter Faktor und wir sind wieder in $O(n)$.

```
void reverse(vector<int> &vec) {
  for (size_t i = 0; i < vec.size() / 2; ++i) {
    size_t other = vec.size() - i - 1;
    int temp = vec[i];
    vec[i] = vec[other];
    vec[other] = temp;
  }
}
```

Die Berechnungen in der Schleife sind konstant und müssen nicht betrachtet werden. Somit reicht es zu wissen wie oft wir die Schleife durchlaufen. Im worst-case müssen wir $\sqrt{n}$ viele Zahlen überprüfen. Damit ist der Algorithmus in $O(\sqrt{n})$.

```
bool is_prime(const uint64_t n) {
  for (uint64_t x = 2; x * x <= n; ++x) {
    if (n % x == 0) {
      return false;
    }
  }
  return n > 1;
}
```

Da wir in jedem Rekursionsschritt $n$ um 1 verringern und nur einmal die Funktion rekursive aufrufen, erhalten wir $n + 1$ aufrufe. Die einzelnen Durchläufe sind konstant und wir erhalten so $O(n)$ als Laufzeit.

```
uint64_t factorial(uint64_t n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Für jeden der $n$ Einträge in dem Vektor müssen wir fib einmal aufrufen. Dabei brauch fib nur konstante Zeit, da die Werte immer aus dem Vektor abgelesen und verwendet werden. Somit erhalten wir eine Laufzeit von $O(n)$.

```
uint64_t fib(uint64_t n, vector<uint64_t> &memo) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else if (memo[n] != 0) // if we already computed the value at n
    return memo[n];
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
void all_fib(uint64_t n) {
  vector<uint64_t> memo(n); // vector contains only zeros
  for (uint64_t i = 0; i < n; i++) {
    cout << fib(i, memo) << " ";
  }
}
```

Die Funktion schreibt uns alle 2er Potenzen kleiner gleich $n$ auf. Damit reicht es zu wissen wie viele es davon gibt, da ein Durchlauf der Funktion konstante Zeit braucht. Da es sich um Potenzen handelt brauchen wir nur $\log(n)$ viele und sind in $O(\log(n))$.

```
uint64_t powers_of_2(uint64_t n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    cout << 1 << " ";
    return 1;
  } else {
    uint64_t prev = powers_of_2(n / 2);
    uint64_t curr = prev * 2;
    cout << curr << " ";
    return curr;
  }
}
```

Es handelt sich nur um eine Berechnung und diese kann in konstanter Zeit umgesetzt werden. Damit sind wir in $O(n)$.

```
uint64_t mod(uint64_t a, uint64_t b) {
  if (b == 0) {
    throw overflow_error("Divide by zero exception!");
  }
  uint64_t div = a / b;
  return a - div * b;
}
```

Für das sortieren brauchen wir $n\log(n)$ Zeit für jeweils einen der beiden Vektoren. Damit haben wir schon mal als Laufzeit $a\log(a)+b\log(b)$. Danach führen wir für jedes Element in $a$ eine logarithmische Suche durch und erhalten zusätzlich $a\log(b)$. Dies ist aber immer kleiner als $a\log(a)+b\log(b)$ und wir haben als Gesamtlaufzeit $O(a\log(a)+b\log(b))$.

```
int intersection(vector<int> &a, vector<int> &b) {
  sort(begin(b), end(b));
  int intersect = 0;
  for (int x : a) {
    if (binary_search(begin(b), end(b), x)) {
      intersect++;
    }
  }
  return intersect;
}
```

2. Example for the following time complexities.
   $O(1)$: Zugriff auf einen festen Index in einem Array.
   $O(\log(n))$: In einem sortierten Array ein Element effizient suchen.
   $O(n)$: In einem unsortierten Array ein Element suchen.
   $O(n\log(n))$: Mergesort zum sortieren eines Arrays.
   $O(n^2)$: Worst-case für Bubblesort zum sortieren eines Arrays.

3. Describe a systematic approach for solving a problem in a programming interview.
   Als erstes ist es wichtig genau zuzuhören, da man sonst besondere Voraussetzungen oder Einschränkungen nicht mitbekommt. Mit diesen ist es gut sich einfache Beispiele zu überlegen und zu schauen wie man diese lösen würde. Hierbei sollte wir auch im Hinterkopf behalten wie Spezialfälle aussehen, die wir beachten sollten. Für das schreiben reicht als erstes eine brute force Methode, die wir im Anschluss optimieren können. So kann man dabei nach überflüssigen schritten, wie doppelter Arbeit, achten um den Algorithmus zu verbessern.

4. Outline the idea of how to retrospectively compute in $O(n)$ the maximum profit when buying and then selling a stock once.
   Zu Anfang wählen wir zwei Variablen min_price_so_far und max_profit. Die Idee ist es in min_price_so_far den kleinsten bis jetzt gesehenen Preis zu speichern. Dies nutzen wir um den potenziellen Gewinn zu berechnen, wobei wir das Maximum in max_profit speichern. Um das Maximum zu finden gehen wir dann durch den Vektor prices durch und aktualisieren min_price_so_far und max_profit in jedem schritt. Am ende müssen wir nur noch max_profit zurückgeben, da dies das Maximum vom gesamten Vektor enthält.

5. How would you check if string b is a permutation of string a.
   Meine Idee ist es in einer Hash-Tabelle zu speichern wie oft einzelne Buchstaben in den jeweiligen Strings vorkommen. Dies reicht, da für Permutationen nur die Anzahl der einzelnen Buchstaben ausschlaggebend ist. Am Anfang fange ich nur den Fall ab, dass die beiden Strings nicht die gleiche Länge haben.

```
int permutation(const string &a, const string &b) {
        if (a.length()!=b.length())return 0;
        map<char, int> dict_a;
        map<char, int> dict_b;
        for (int index = 0; index < a.length(); index++){
        dict_a[a[index]]++;
        dict_b[b[index]]++;
        }
        return (dict_a == dict_b);
}
```

# Bit Manipulation Primitive Types and Arrays

1. How can we exploit bit operations for operations with sets?
   Mengen, wo wir genau wissen welche Elemente vorkommen, können wir als einen Bitstring darstellen. Dabei entspricht jedes Bit einem Element und gibt an ob dies in der Menge enthalten ist. So übersetzen wir eine Menge in eine Zahl mit der wir einfacher und schneller arbeiten können. Dabei entsprechen Mengenoperationen wie durchschnitt oder Vereinigung dem bitweise and oder or. Auch alle weiteren einfachen Mengenoperationen wie Komplement ($\sim a$) oder Differenz ($a|\sim b$) können wir so darstellen.

2. Explain the difference between arithmetic and logical right shifts for negative signed integers.
   Bei einem logical right shift wird der ganze Integer samt dem Vorzeichenbit verschoben. Dies kann dazu führen, dass eine negative Zahl positive wird da das Vorzeichenbit verschoben und durch eine 0 ersetzt wird.
   Arithmetic right shifts hingegen füllt die neuen stellen mit dem Wert des Vorzeichenbits. So bleibt das Vorzeichen erhalten und es entspricht ungefähr einem durch 2 teilen.

3. Explain what the following code does $(n\&(n-1)) == 0$.
   Dazu stellen wir als erstes fest, dass der Ausdruck $(n\&(n-1)$ die erste 1 von rechts entfernt. Angenommen diese 1 ist an position $x$, dann sind rechts von $x$ nur Nullen. Damit sehen wir, dass $n$ und $n-1$ sich auf $x$ und allen Bits rechts davon unterscheiden und beim logischen und zu 0 werden. Mit diesen Überlegungen wissen wir, dass $(n\&(n-1)) == 0$ nur gelten kann, wenn $n$ nur ein von 0 verschiedenes Bit haben kann, da sonst das zweite Bit von rechts unverändert bleibt. Dies ist aber genau dann der Fall, wenn $n$ eine 2er Potenz ist. Damit prüft $(n\&(n-1)) == 0$ ob es sich um eine 2er Potenz handelt.

4. Outline an algorithm to count the number of bits that are set to 1 in an integer.
   Wie oben gezeigt entfernt $n\&(n-1)$ immer die erste 1 von rechts. Somit können wir diese Zeile in einer while-Schleife solange durchführen bis $n == 0$ gilt. In jedem Durchlauf entfernen wir dabei eine Eins und erhalten so mit einem Zähler wie viele im ursprünglichen $n$ enthalten waren.

5. Outline an algorithm to reverse the bits in an unsigned integer.
   Dafür gehen wir mit einer for-Schleife die $n$ Stellen von $a$ durch und speichern das Zwischenergebnis in der Variable result. Diese aktualisieren wir jeden Durchlauf mit:
   $result = result|(((a\&2^i) >> iu) << (n-i)u)$
   Dabei gibt uns $(a\&2^i)$ das Bit von der i-ten Stelle und mit den anderen beiden Teilen verschieben wir dies auf den richtigen Platz.

6. Imagine you have to reorder integers in an array so that the odd entries appear first.
   How is this problem related to Quicksort's partitioning function?
   In dem Algorithmus gehe ich durch den ganzen Vektor durch und merke mir immer wo die nächste ungerade Zahl hingeschrieben werden muss. Wenn eine gefunden wird, wird werden die Einträge an beiden stellen getauscht und so stehen am ende alle ungeraden Einträge am Anfang.

```
void odd_order(vector<int> &v) {
    size_t write_idx = 0;
    for (size_t i = 0; i < v.size(); ++i) {
        if (v[i]%2) {
            std::swap(v[write_idx++],v[i]);
        }
    }
}
```

   Dies ist ähnlich zu Quicksort, da wir dort an stelle aller ungeraden Zahlen alle Zahlen, die kleiner als das Pivotelement sind haben wollen. So können wir die Funktion hier dementsprechend anpassen, indem $v[i]\%2$ durch $v[i] < p$ ersetzt wird. Zusätzlich muss nur noch der Zähler ausgegeben werden, damit wir wissen wo die Grenze zwischen den beiden Bereichen ist.

# Arrays and Strings

1. Outline an algorithm to convert a string to an unsigned integer
   Ich gehe davon aus, dass auch die Zahl im String positiv ist. Zum umwandeln erstelle ich als erstes eine Variable *num* mit dem Wert 0. Diese Speichert den Wert der bereits eingelesenen Zahl. Dann

gehe ich mit einer for-Schleife von links nach rechts durch den string. Dabei wird in jedem schritt $num = 10 * num + (s[i] +' 0')$ ausgeführt um $num$ zu aktualisieren. Dies funktioniert da $s[i] -' 0'$ dem Integer Wert der Ziffer entspricht. Zum Schluss gebe ich noch $num$ zurück.

2. Outline an algorithm to convert an unsigned integer to a string.
In der Variable $s$ speichere ich die derzeitige Form vom String. In der do-while-Schleife nehme ich mit $char(num\%10) +' 0'$ immer die letzte Stelle und wandle sie in den entsprechenden character um. Danach entferne ich mit $num/ = 10$ die letzte Stelle vom Integer $num$. Der String $s$ ist jetzt noch in der falschen Reihenfolge und deswegen wird er noch in der for-Schleife invertiert.

```
inline string int_to_string(int num) {
  string s;
  do {
    s += char(num%10)+'0';
    num /= 10;
  } while (num);
  for (size_t i = 0; i < s.size() / 2; ++i) {
    swap(s[i], s[s.size() - 1 - i]);
  }
  return s;
}
```

3. Explain the characteristics of the character encodings
ASCII, Extended ASCII, UTF-8, UTF-16 and UTF-32
Bei allen Formaten handelt es sich um Kodierungen für bestimmte Symbole. Die Unterschiede liegen darin wie viele unterschiedliche Symbole codiert werden können und dabei kommt es bei manchen noch zu weiteren Besonderheiten.
ASCII: Es werden 128 Zeichen codiert, was vergleichsweise wenig ist und somit nur wenig Platz für seine 8-Bit (das erste ist immer 0) pro Zeichen brauchen.
Extended ASCII: Hier wird zusätzlich zu ASCII noch das erste Bit benutzt und wir haben 256 Zeichen.
UTF-8: Ist eine dynamische Kodierung, d.h. wir codieren in einem Präfix wie die folgenden Bits zu lesen sind. Damit können sowohl normale ASCII Symbole als auch viele weitere Dargestellt werden.
UTF-16: Hier werden Zeichen entweder mit einem oder zwei Wörtern codiert, dabei ist ein Wort 16 Bit lang. Damit hat dies eine variable-size.
UTF-32: Hier werden alle Symbole mit 4 Bytes gespeichert unter anderem auch ASCII Symbole. Deswegen ist es fixed-size.
Die Vorteile von zum Beispiel UTF-8 sind, dass es weniger Speicher für zum Beispiel normale ASCII Symbole brauch. UTF-32 hingegen hat den Vorteil, dass es leicht ist zu sehen wo ein Zeichen anfängt und aufhört. So ist dies bei UTF-8 nicht ohne Berechnung des Präfixes möglich.

4. What advice can you give to clean up strings as quickly as possible?
Es sollte alles möglichst in-place passieren, da einzelne character aus einem String entfernen lange braucht. Deswegen sollte man sich auch immer sicher sein ob man dies eventuell auch weglassen oder am ende in einem einzigen Schritt machen kann. So ist es möglich in Zwischenschritten nur white-space Zeichen hinzuzufügen und diese erst ganz am ende zu entfernen.

5. Why are hardcoded regex expressions often much faster?
Regex expressions sind sehr langsam, da diese sehr allgemein funktionieren müssen. So werden diese Ausdrücke zur Laufzeit des Programmes in state machines compiliert. Dies mach sie deutlich langsamer im Vergleich zum hardcoden eines einzelnen Ausdrucks.

6. Is run-length encoding a good compression method for strings?
Run-length ist effektive, wenn es sich um viele sich wiederholende Zeichen handelt. So verändert sich bei jeweils nur doppelt vorkommenden Zeichen nichts aber ab längeren gleichen Symbolketten sparen wir Platz. In einem normalen String für Text kommen aber nur selten sich wiederholende Zeichen vor und wir kommen zum Problem, dass einzelne Zeichen mit zwei stellen codiert werden, was Speicherintensiver ist. Somit ist run-length für Text keine gute Idee, da der String dabei nur länger wird.

## Linked Lists Stacks Queues BTs and Tries

1. How to delete a (non-tail) node from a singly linked list without knowing its predecessor in O(1) time?

Als erstes kopieren wir den Inhalt von dem Nachfolgerknoten in den, den wir löschen möchten. Dazu gehört auch der Pointer vom Nachfolger. Somit wurde der zu löschende Knoten vollständig überschrieben und ist nicht mehr in der Liste enthalten. Da wir auch den Pointer übertragen haben zeigt dieser Knoten wieder auf den ursprünglichen Nachfolger vom Nachfolger.

2. Explain the runner technique using a linked list example.
Bei der runner technique benutzt man zwei Pointer die parallel durch die linked list gehen. Dabei geht einer in jedem Schritt einen Knoten weiter und der andere immer zwei Knoten. Eine Anwendung wäre es die Hälfte einer linked list zu finden. So kann man mit beiden durch die liste durchgehen und falls der schnelle Pointer ans Ende gelangt ist, wissen wir dass der Langsame in der Mitte von der Liste ist. Diese Methode ist deutlich schnelle als ein naiver Ansatz mit einem Pointer, da wir nur einmal durch die linked list gehen müssen.

3. If you were to tune the code for speed, how would you implement a stack?
Um einen stack zu implementieren bietet es sich an einen dynamisch veränderbaren Vektor zum speichern zu nehmen. Die Funktionen $push\_back(), empty(), back()$ und $pop\_back()$ setzen dabei alles um was wir für den stack brauchen.

4. How can we use a std::list in C++ as a queue?
Bei einer queue brauchen wir die Funktionen *enqueue*, *dequeue*, *size* und *empty*. Dabei können wir für *size* und *empty* die Standardfunktionen von einer Liste verwenden. Für *enqueue* benutzen wir $emplace\_back()$. Bei *dequeue* hingegen speichern wir als erstes wir das erste Element und entfernen es dann mit $pop\_front()$. Dieses Element geben wir dann noch zurück damit es sich wie *dequeue* verhält.

5. How does a binary tree and a binary search tree differ?
Ein binary tree ist ein Baum in dem jeder Knoten maximal 2 Kinder hat. Dabei gibt es aber keine weiteren Einschränkungen. Ein binary search tree hingegen ist ein binary tree mit der Eigenschaft, dass das Maximum vom linken Teilbaum kleiner gleich und das Minimum vom rechten Teilbaum echt größer als die Wurzel ist.

6. Describe the different types of binary tree traversals.
Es gibt drei verschiedene arten einen binary tree durchzugehen. Diese werden klassischer Weise rekursive definiert, da dies schneller berechnet wird. Im folgenden gebe ich an in welcher Reihenfolge die Wurzle und die jeweiligen Teilbäume rekursiv durchgegangen werden.
Preorder: Wurzel, linker Teilbaum, rechter Teilbaum. Eignet sich zum kopieren des Baumes.
Postorder: linker Teilbaum, rechter Teilbaum, Wurzel. Eignet sich zum löschen eines Teilbaumes.
Inorder: linker Teilbaum, Wurzel, rechter Teilbaum. Falls es sich um einen search tree handelt, erhalten wir eine sortierte Liste aller Werte.

7. How does a trie and a radix tree differ?
Ein radix tree ist ein trie, den wir komprimiert haben. Dabei ist ein radix tree ein trie, in dem wir alle Ketten von Knoten mit jeweils nur einem Kinderknoten zu einem einzigen zusammenfassen. Damit enthalten die Knoten in einem radix tree Wörter im Gegensatz zu ein trie wo es nur einzelne Characters sind.

## Heaps, Sorting and Hash Tables

1. Describe briefly the heap data structure.
Ein Heap ist ein vollständiger binär Baum, wobei die Blätter von rechts nach links befüllt werden (shape property). Mit dieser Einschränkung ist es möglich einen Heap als Array zu speichern, da man aus der Position im Array eindeutig die Position im binär Baum berechnen kann. Weiter muss noch gelten, dass der Schlüssel an jedem Knoten größer gleich (max heap) als der Schlüssel der beiden Kinderknoten ist (heap property).

2. How would you find the k longest words in a data stream?
Ich würde einen min heap benutzen. Dabei werden als erstes die ersten k Wörter hinzugefügt. Danach vergleiche ich das neue Wort immer mit dem min vom heap (konst Zeit) und füge es hinzu, wenn es sich um ein längeres Wort handelt. Damit der Heap danach immer noch nur k Elemente enthält entferne ich das kleinste Element. Diese beiden schritte können auch verbunden werden, um es etwas schneller zu machen. Am ende gebe ich dann alle k Elemente aus dem Heap aus, da diese nach Konstruktion die k längsten Wörter aus dem data stream sind.

3. How would you address the problem of merging multiple sorted files that are too large for RAM?
   Dazu bietet sich wieder ein min heap an. Dabei arbeiten wir aber mit verweisen auf die Dateien und merken uns wo wir in der jeweiligen Datei sind. Der heap ist dabei nach der Größe von diesen jeweiligen Elementen sortiert. So können wir immer nach dem kleinsten Element fragen und dieses Element der großen Liste hinzufügen. Danach gehen wir in der jeweiligen Datei ein Element weiter und falls es nicht leer ist, fügen wir es wieder zum heap hinzu.

4. What are sorting networks?
   Ein sorting network besteht aus mehreren COEX (compare and exchange) Teilen. Wie der Name sagt, vergleichen diese immer zwei Elemente und vertauschen diese Falls sie nicht sortiert sind. In einem sorting network benutzt man diese um so jede Eingabe zu sortieren, da man gezielt die Elemente an bestimmten Positionen vergleicht und gegebenenfalls tauscht. Dies entspricht einem hardcoden vom sortieren.

5. Why sorting may speed up set operations?
   Mit sortieren Arrays lassen sich bestimmte Operationen wie das suchen von Elementen deutlich beschleunigen, da man nicht potentiell den ganzen Array durchgehen muss. Weiter kann man Operationen wie Schnitte leichter berechnen, da man so beide Arrays parallel durchgehen kann und nicht nach bestimmten Elementen extra suchen muss.

6. What is a minimal perfect hash?
   Eine Hashfunktion, die gegebene Wörter auf eine Hashtabelle abbildet, ist minimal perfekt, falls es keine Kollisionen (perfect) oder leere Felder (minimal) in der Hashtabelle gibt.

## Backtracking, Divide-and-Conquer, Dynamic Programming, and Greedy Algorithms

1. What is backtracking?
   Hierbei handelt es sich um eine Programmierstrategie. Dabei werden in jedem Schritt alle gültigen nächsten Schritte berechnet und ausprobiert. Man geht dabei, falls es keine weiteren gültigen Schritte mehr gibt, im Suchbaum zurück.
   Man testet so alle erlaubten möglichen Schritte durch und findet damit die Lösung. Damit entspricht backtracking einer vollständigen Tiefensuche.

2. How would you proceed to compute the n-Queens problem with backtracking as fast as possible?
   Da in jeder Zeile des Schachbrettes immer nur eine Dame sein kann, können wir es mit einer einfachen Liste realisieren, wobei der Eintrag die Spalte angibt. Dann laufen wir die Listeneinträge durch und gehen in alle möglichen Spaltennummern, die eingegeben werden können.
   Falls wir die Liste gefüllt haben, haben wir eine zulässige Belegung gefunden und machen weiter. Falls es aber keine weitere Möglichkeit für die Spalte gibt, so können wir eine Zeile zurück gehen und dort weitere Spaltenpositionen probieren.

3. Describe some ideas how to solve the 15-puzzle with backtracking.
   Als erstes brauchen wir eine Liste aller bisher besuchten Felder um unendliche Ketten zu vermeiden. Beim berechnen des nächsten Zuges, überprüfen wir dann immer ob dieser bereits betrachtet wurde. So ist es dann möglich, dass wir keine weiteren möglichen Züge mehr haben und im Backtracking einen Schritt zurück gehen müssen.
   Um den Algorithmus etwas besser zu gestalten können wir noch eine Heuristik für gute Züge einfügen.

4. Why is efficient parallelization not trivial in most divide and conquer algorithms
   Für eine effiziente Implementierung muss man alles parallelisieren, dies ist aber bei den ersten Anwendungen von divide nur schwer möglich. So geht man da klassischer weise immer durch einen großen Teil der Daten durch. Es wäre aber nicht wirklich effizient zu erst einige normale Schritte zu machen und erst später alles zu parallel auszuführen.

5. Why may decrease and conquer be a better name choice for algorithms like quickselect or recursive binary search than calling them divide and conquer?
   Der große Unterschied ist, dass man bei normalem divide and conquer, in beide Teile vom divide geht. Dies ist bei quickselect nicht der Fall, da wir dort ausschließlich einen betrachten. Damit verringern wir das Problem eher als es aufzuteilen. Insbesondere wird die Anzahl an betrachteten Daten kleiner, im Gegensatz zu normalem divide and conquer.

6. What is dynamic programming?
   Dies ist ähnlich zu divide and conquer, da wir wieder die Lösung auf ein kleineres Problem reduzieren. Insbesondere achten wir darauf, die kleineren Teilprobleme immer nur einmal zu lösen, um das Programm effizienter zu machen.
   Dies können wir erreichen, indem wir alle Zwischenergebnisse abspeichern und nur noch nachschauen müssen, ob diese bereits gelöst wurden.

7. Compare the top-down approach with the bottom-up approach used in dynamic programming.
   Bottum-up entspricht einem iterativem Programm wobei der cash kleiner ist. Dies kommt daher, dass wir bei einem leichteren Problem anfangen und danach die schwirigeren betrachten.
   Top-down hingegen entspricht rekursivem Programmen und kann somit leichter implementieren zu sein. Hier gehen wir rekursive von schweren Problemen zu einfacheren über. Diesen Ansatz kann man leichter optimieren, da wir bestimmte Teilprobleme gar nicht erst berechnen, wenn diese nicht Auftauchen.

8. Present briefly a greedy heuristic of your choice.
   Bei einer greedy Heuristik schaut man sich lokal das beste Ergebnis an.
   Als Beispiel könnte man nach dem Maximum einer Funktion suchen und als greedy Heuristik die Richtung nehmen in die die Funktion ansteigt. Dies gibt uns immer ein lokales Maximum aber es kann noch weitere geben, die dieses Programm nicht findet, da es dafür erstmal runter gehen muss.

## Compilers

1. Describe the steps necessary to get a syntax tree from a grammar.
   Als erstes bekommen wir eine Eingabe und formen diese mit einem Scanner in Terminalsymbole von unserer Grammatik um. Der Parser formt dies dann in Bestandteile der Grammatik um und überprüft gleichzeitig ob dies korrekt ist.
   Dies wird daraufhin zu einem Syntax tree zusammengesetzt. Zweistellige Operationen werden dabei dargestellt, indem wir diese in die Wurzel schreiben und die anderen Bestandteile in die Kinder schreiben. Einstellige Operationen machen wir ähnlich nur, dass wir dort nur ins linke Kind schreiben. Mit diesen regeln können wir sukzessive unseren AST aufbauen.

2. What is the task of a scanner?
   Der Scanner bekommt eine Eingabe und formt diese in ein Symbol um. Symbole bestehen dabei aus einer Beschreibung und einem Wert. Der Wert muss dabei ein Terminal sein. So werden zum Beispiel mehrere aufeinander folgende Zahlen zu einer zusammengesetzt.

3. Describe the parser function for the rule d={'a'}['b']'c'.
   Als erstes dürfen beliebig viele (einschließlich 0) a's hingeschrieben werden. Dies können wir überprüfen, indem wir die mit einer while-Schleife jedes a durchgehen.
   Danach ist ein einziges optionales b möglich, was wir mit einem if überprüfen können.
   Zum schluss müssen wir noch testen ob das letze Zeichen ein c ist, was wir wieder mit einem if machen können.

4. How can a syntax tree be optimized?
   Eine erste Optimierung wäre es einfache Rechnungen wie + oder - direkt auszuführen, falls in beiden Blättern nur Konstanten stehen. Dies kann man auch auf - bei negativen Zahlen oder anderes erweitern und so das Ausführen vom Syntax tree beschleunigen.
   Andererseits kann man nach gleichen Teilbäumen suchen und diese nur einmal berechnen, um Redundanz zu verringern. Oder es können bestimmte Rechnungen einfach übersprungen werden, wenn sich das Ergebnis nicht mehr verändern kann, wie wenn man etwas mit 0 multipliziert.

## Programm Competition: Code a la mode

1. Describe how your bots decides which action it will do next.
   Am Anfang jeder Runde wird als erstes getestet, ob ein Gericht zusammengestellt und abgegeben werden kann. Dies besitzt die höchste Priorität und wird sofort erledigt. Danach wird geschaut was mit möglichst wenig Schritten erledigt werden kann und er kümmert sich um die Zubereitung von diesen fehlenden Zutaten.

2. Are there situations where your bot behaves in a sub-optimal way?
   Die Wege zum einsammeln sind sehr ineffizient gemacht. So wird nicht auf die Position von Zutaten wie Blaubeeren oder Eis geachtet, sondern immer in der gleichen Reihenfolge alles eingesammelt. Weiter wird nicht auf die Position vom anderen Spieler geachtet was manchmal zu sehr vielen sinnlosen Schritten führt.

3. What are some heuristics to improve your bots performance?
   Eine der wichtigsten Verbesserungen war es beim Gegenstände ablegen immer das nächste leere Feld zu nehmen. Damit haben sich die Zeiten für das Bewegen drastisch verkürzt.
   Dies könnte man jetzt auch für die Reihenfolge beim Aufsammeln von Zutaten anwenden. Dabei kann deren Position und auch die Position vom anderen Spieler berücksichtigt werden um alles mit möglichst wenig schritten zu erreichen.
   Eine andere Möglichkeit wäre es zu zu berücksichtigen, was der andere Spieler mit dem Ofen macht, beziehungsweise was bereits im Ofen ist.
   Beides sollte eine gewisse Verbesserung mit sich bringen, da redundante Züge vermieden werden und das jetzige Programm auf nichts in dieser Art achtet.

## Graph Traversal and Shortest Paths

1. If you need to find the shortest path between two nodes in a graph where all edges have length 1, which algorithm would you use and why?
   Das Problem entspricht kürzestem weg in einem ungewichtetem Graphen. Damit können wir eine Breitensuche machen, da diese die Knoten der Entfernung vom Startknoten aus durchgeht. Diese ist damit auch am effizientesten, da wir ohne eine Heuristik immer alle Knoten mit der gleichen Entfernung durchgehen müssen.

2. Assume that you need to find a shortest path in a rel. small graph with arbitrary edge lengths, and you have to have a working algorithm fast. Which algorithm would you choose and why?
   Hier bietet sich der Floyd-Warshall Algorithmus an, da dieser am einfachsten zu implementieren ist und wir auch keine Probleme bei negativen Kantengewichten bekommen.
   Die Laufzeit unterscheidet sich bei wenigen Knoten kaum von Dijkstra und wir haben somit kaum Einbußen in der Laufzeit.

3. Why does Dijkstra's algorithm not work with negative edge lengths?
   Dijkstra bricht die Suche von einem Pfad ab, falls dieser teurer wird als ein bereits gefundener Pfad. Falls es aber negative Kanten gibt, ist es möglich, dass damit der Pfad danach wieder günstiger wird. Somit kann bei einem Pfad mit hohen Anfangskosten aber geringen Gesamtkosten dazu kommen, dass Dijkstra den nicht benutzt sondern einen anderen weniger guten Pfand nimmt.

## Spanning Trees and Topological Sorting

1. Argue why topological sortings only exist for acyclic graphs.
   Angenommen es gibt einen Zykel, dann gibt es zwei Knoten A und B mit der Eigenschaft, dass es einen Pfad von A nach B und von B nach A gibt. Damit muss A hinter B und B hinter A in der topologischen Sortierung stehen, damit die Eigenschaft erfüllt ist. Dies ist aber ein Widerspruch, da es solch eine Sortierung nicht geben kann.

2. Briefly describe how you approached and solved Project Euler Problem 1.
   Es bietet sich eine einfache for-schleife an, wo wir durch alle Zahlen durchgehen und überprüfen ob es sich um einen Teiler handelt. Dann müssen wir es nur gegebenenfalls auf unsere Summe addieren.

3. Describe your approach on Project Euler Problem 107: which algorithm did you use and why?
   Dazu bauen wir einen minimalen spanning tree mit Prim's Algorithmus. Wir haben nicht Kruskal's Algorithmus genommen, da man dort noch Kreisfreiheit prüfen müsste. Da insbesondere nur die Kantensumme gesucht wird können wir den Algorithmus noch ein bisschen vereinfachen. So kann bereits beim einlesen der Adjazenzmatrix bereits die Summe berechnet werden.

# Maximum Flow and Minimum Cut

1. Describe the general idea of the algorithm by Edmonds/Karp in your own words.
   Beim dem Algorithmus geht es darum den maximalen Fluss in einem Graphen zu berechnen. Dafür starten wir mit einem beliebigen Pfad zwischen dem Start- und Zielknoten. Damit haben wir bereits unseren ersten Fluss gefunden und fügen jetzt noch die hinzu, die wir durch andere Pfade bekommen. Dazu suchen wir mit Breitensuche nach allen anderen Pfaden vom Start- zum Zielknoten. Um ungünstige erste Entscheidungen später noch ausgleichen zu können führen wir Rückwärtskanten ein. Diese Laufen immer in die entgegengesetzte Richtung. Falls ein neuer Fluss über so eine Kante geht, wissen wir, dass wir mehr Fluss erzeugen können, wenn wir weniger Fluss über diese Kante schicken.

2. Describe your algorithm that solves the Problem 100 ("3n+1-problem")
   Die Hauptidee ist es bei allen Zahlen die Anzahl rekursiv zu berechnen und die Zwischenergebnisse abzuspeichern. Mit dem Speichern von den Zwischenergebnissen erspart man sich redundante Berechnungen. Weiter wird die Berechnung abgebrochen, sobald einer 2er Potenz gefunden wurde, da es in diesem Fall nur noch $\log_2(n)$ viele Schritte bis zur 1 braucht. Zum Schluss müssen wir nur das Maximum aktualisieren.

3. Which algorithm do you use for Problem 459 ("Graph Connectivity"), and why?
   Wir müssen uns alle Knoten und Kanten in einem zusammenhängenden Teilgraphen anschauen um zu überprüfen ob wir alle Verbundenen Knoten gefunden haben. Damit wissen wir auch, dass eine Breitensuche bereits optimal ist, da diese genau das macht.
   Für die Breitensuche können wir uns einen beliebigen Knoten nehmen und dann alle verbundenen Knoten finden. Falls wir damit fertig sind, suchen wir uns einen neuen Knoten und erhöhen den Zähler um Eins. Dies machen wir so lange bis es keine unbesuchten Knoten mehr gibt.

# Maximum Flow and Minimum Cut

1. n-to-m-assignment problem:
   Wir nehmen den gegebenen Graphen für die Qualifikationen und fügen noch zwei Knoten s und t hinzu. Die gewichte vom Graphen für die Qualifikationen werden immer auf 1 gesetzt. Als nächstes müssen wir die Knoten s und t mit dem Graphen verbinden. Dazu wird s mit allen Arbeitern verbunden, wobei die Kante als Gewicht deren Kapazität bekommt. Der Knoten t wird mit allen Aufgaben verbunden, wobei deren Kanten die Kosten als Gewicht bekommen.
   Auf diesen Graphen wenden wir unsere MaxFlow black-box an. Die Menge an Fluss von einem Arbeiter zu einer Aufgabe gibt nun an wie viel jeder Arbeiter von dieser Aufgabe übernehmen muss. Input: $G = (W \cup T, E)$

```
for k in E:
  k bekommt Gewicht 1
V=V+{s,t}
for w in W:
  E.add((s,w,w.capacity))
for a in T:
  E.add((t,a,a.cost))
# benutze maxFlow
Flow = maximum_flow(V,E,s,t)
# weise Arbeit zu
for w in W:
  for edge in Flow.edges:
    w.assign(edge.end_node)
```

2. Describe the MinCut-Problem and give a simple example
   Ein Schnitt ist eine Teilmenge von Kanten, die von einem Graphen entfernt werden. Beim minimalen Schnitt geht es darum einen Schnitt zu finden, sodass der restliche Graph unzusammenhängend, die Knoten s und t in unterschiedlichen Komponenten und die Summe der Gewichte von den entfernten Kanten minimal ist.

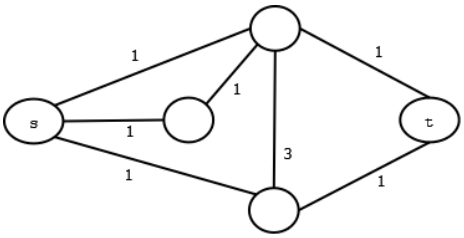


Abbildung 1: Genau ein minimaler Schnitt

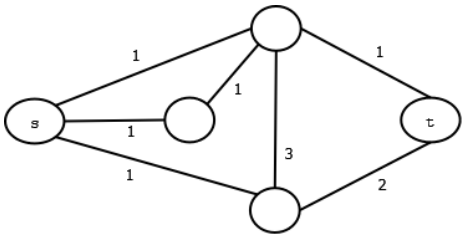


Abbildung 2: Genau drei minimale Schnitte

3. Describe the Matching-Problem and give a simple example
   Sei G ein bipatiter Graph. Dann ist ein matching eine Teilmenge von Kanten wobei jeder Knoten nur einmal vorkommt. Ein perfektes matching ist dabei ein matching wobei jeder Knoten zu genau einer Kante gehört.

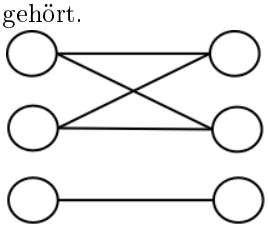

Abbildung 1: Genau zwei perfekte matchings

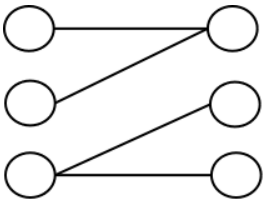

Abbildung 2: keine perfekten matchings

# CodeGolf

1. Describe one change that significantly decreased your program length.
   Eine sehr große Ersparnis wurde durch das Wiederverwenden von Variablen erreicht. So wurde auch nicht verwendeter Input in immer die gleiche Variable geschrieben. Weiter gab es auch später neuen Input, wo die Variablen wieder verwendet werden konnten.
   Wir konnten so ca. 60 zeichen sparen.

2. shortest path between two specified nodes in a graph.
   Hierfür bietet sich der Floid-Warshall Algorithmus an, da dieser besonders kurz ist. Die Laufzeit ist zwar schlechter als bei Dijkstra aber dies ist beim Codegolf unwichtig.