Aufgabe 3.1 - Samplesort

$A_=[20, 32, 86, 75, 51, 42, 16, 29, 62, 13, 99, 82, 73, 39, 21, 8, 79, 31]$
Schritt 1: Verteilen der 18 Schlüssel auf 3 Rechner:
Rechner 1: $A_1 = [20, 32, 86, 75, 51, 42]$
Rechner 2: $A_2 = [16, 29, 62, 13, 99, 82]$
Rechner 3: $A_3 = [73, 39, 21, 8, 79, 31]$
Jeder Rechner sortiert nun seine Schlüssel mit Quicksort und bestimmt die Schlüssel an Positionen $i \cdot n/p^2; i \in [0..(p-1)]$ Also Positionen 0, 2 und 4.
Rechner 1: $A_1 = [\mathbf{20}$,32,$\mathbf{42}$,51,$\mathbf{75}$,86] $\rightarrow$ Schlüssel: 20, 42, 75
Rechner 2: $A_2 = [\mathbf{13}$,16,$\mathbf{29}$,62,$\mathbf{82}$,99] $\rightarrow$ Schlüssel: 13,29,82
Rechner 3: $A_3 = [\mathbf{8}$,21,$\mathbf{31}$,39,$\mathbf{73}$,79] $\rightarrow$ Schlüssel: 8,31,73
Rechner 1 erhält $p^2 = 9$ Schlüssel, sortiert und versendet dann die an Positionen $i \cdot p \; ; i \in [1..(p-1)]$, Also Positionen 3 und 6: $S_1 =$ [8,13,20,$\mathbf{29}$,31,42,$\mathbf{73}$,75,82].
Jeder rechnet teilt nun gemäß S = [29,73] seine Folge in p=3 Partitionen. Wobei in der i-ten Partition alles größer gleich $Pivot_{i-1}$ und echt kleiner $Pivot_i$ ist.
Hier also : $I_1 =]-\infty, 29[$; $I_2 = [29, 73[$; $I_3 = [73, \infty[$
Rechner 1: $A_1 =$ [(20) , (32,42,51) , (75,86)]
(32,42,51) an Rechner2; (75,86) an Rechner 3
Rechner 2: $A_2 =$ [(13,16) , (29,62) , (82,99)]
(13,16) an Rechner 1; (82,99) an Rechner 3
Rechner 3: $A_3 =$ [(8,21) , (31,39) , (73,79)]
(8,21) an Rechner 1; (31,39) an Rechner 2
Es wird also $[8, 13, 16, 20, 21][29, 31, 32, 39, 42, 51, 62][73, 75, 79, 82, 86, 99]$
zusammengesetzt zu: [8,13,16,20,21,29,31,32,39,42,51,62,73,75,79,82,86,99].

3.2 - k Teilfolgen

Da jede Teilfolge an sich sortiert ist, können diese nach der Idee von Mergesort zusammengemischt werden. Bei k Teilfolgen wären aber für das Minimum aus den Teilfolgen k Vergleiche nötig. Die Idee ist nun die jeweils kleinsten Elemente der sortierten Teilfolgen in einen Min-Heap zu packen und diesen zu reparieren. Dann das Minimum herauszunehmen und aus der entsprechenden Teilfolge das nächstgrößere wieder in den min-Heap einzufügen (inkl. Reperatur). Damit hat der Min-Heap immer nur k Elemente und es wird für die Reparatur immer nur log(k) benötigt. Dies für n Elemente ergibt Laufzeit (n log(k)).

Um zu wissen, aus welcher Teilfolge das jeweils entfernte kommt nutzen wir einen "MinPairHeap ", der Paare speichert, aber nur nach der ersten Paar-Komponente sortiert.
Pseudocode:

```
// Die Teilfolgen seien adressierbar durch die Arrays
// A.1, A.2, A.3... A.k
// initialisiere
Int tempmin,templist , u; Tempin
Array Ziel[n] // Zielarray
Array Pos[k] // die jeweilige Position im Array
for (i=0 ; i<k ;i++) Pos[i] = 0; // startpositionen
MinPairHeap H1; // insert hat repair inkludiert
for (i=0 ; i<k ;i++) {
        H1.insert( (A.i[0],i) ) // aus jeder Min einfuegen,
}                               // mit listenverweis.
while u<n:{
  (tempmin,templist) = H1.delete_min()
  if (1+Pos[templist] < len(A.templist) ){
    Pos[templist]++       // listenposition eins erhoehen
    H1.insert(A.templist[ Pos[templist],templist ] )
  }
  Ziel[u] = tempmin
  u++
}
```

Die Laufzeit lässt sich leicht erkennen. Zu Beginn ist eine for schleife mit $O(k)$, dann durchlaufen wir exakt n Mal die while Schleife. In der While-Schleife führen wir in der insert funktion $O(log\ k)$ Operationen durch. Damit bleiben wir in $O(n \cdot log(k))$.
Die Korrektheit lässt sich durch die sortiertheit der Teilfolgen begründen. Da immer die jeweilig kleinsten Elemente in den Heap eingefügt werden, muss das kleinste noch nicht im sortierten Array liegende Element auf jeden Fall im Heap liegen. Dieses wird dann ins Ziel geschrieben. Somit wird in jedem Durchlauf garantiert das Minimum der noch nicht sortierten gewählt.

3.3 - Fast sortiert

Für ein beliebig sortiertes Array gibt es n! Ordnungstypen oder auch Permutationen. Auf einem fast sortierten Array kann ich um eine bestimmte Position in Abstand k (k/2 nach links, k/2 nach rechts) genau k Elemente verteilen. Wir wählen also k-elementige Teilfolgen. Diese sind durch die Wahl der Position beschränkt. Für diese Elemente gibt es k! mögliche Anordnungen und ich kann genau n/k solche Gruppen bilden. Da jede verschiedene Anordnung eine andere Permutation ergibt, folgen $k! \cdot k! \cdot k! \cdot ..... \cdot k!$ mögliche Permutationen, mit genau n/k Faktoren, also $(k!)^{n/k}$ verschiedene.
b)
Analog zum Beweis aus der Vorlesung können wir zeigen, das die worst Case Laufzeit immer mindestens der maximalen Tiefe des (Vergleichsbasierten-)Sortierbaums entspricht. Dieser ist für ein vergleichsbasiertes Verfahren auf einem fast sortierten Array mit Parameter k ein möglichst vollständiger Binärbaum mit $(k!)^{n/k}$ Blättern, wie wir in der a) gezeigt haben. Die Tiefe dieses Baums ist dementsprechend $log_2((k!)^{n/k}) = n/k \cdot log_2(k!)$. Wir wissen, dass $\Theta(log(k!)) = \Theta(k \cdot log(k))$. Damit ergibt sich für die worst Case Laufzeit eben $\Omega(n/k \cdot k \cdot log(k)) = \Omega(n \cdot log(k))$
c)
Für den Allgemeinen Beweis betrachten wir die möglichen Permutationen des Arrays. Wir können die Anordnung des Arrays durch eine sukzessive Verteilung der n Elemente beginnend bei 1 auf die jeweiligen Plätze darstellen. Für ein allgemeines Array gilt wie in der Vorlesung, das erste Element hat n mögliche Plätze, das zweite n-1, usw. Diese Möglichkeiten multipliziert ergeben genau die n! Permutationen.
Für ein mit Parameter k fast sortiertes Array gibt es für die 1 genau k+1 Plätze beim verteilen. Für die 2 ständen k+2 Plätze zur Verfügung, aber auf einem davon liegt auf jeden Fall die 1. Analog gilt es für 3,4 usw. Jedes dieser i-ten Elemente hat k+1 mögliche Plätze. Das gilt bis zu i+k = n. Das darauffolgende hat nur noch die letzten k Plätze, das danach nur noch k-1, dann k-2 bis zum letzten Element, das durch die Wahlen zuvor festgelegt ist. Damit gibt es also
für die ersten n-k Elemente jeweils k+1 Möglichkeiten, das ergibt $(k+1)^{n-k}$ Permutationen und für die folgenden gerade $k \cdot (k-1) \cdot (k-2) \cdot ... \cdot 1 = k!$
Die Anzahl der Permutationen ist also $(k+1)^{n-k} \cdot k!$.
Das ist die Anzahl der Blätter in dem entsprechenden Ordnungsbaum. Ein vergleichsbasierter Algorithmus besitzt als Sortierbaum immer einen Binärbaum. Die Tiefe eines Binärbaums mit eben sovielen Blättern ist dann
$log_2((k+1)^{n-k}) \cdot k!)$ Damit ist die Tiefe in
$\Theta(log(k! \cdot (k+1)^{n-k})) = \Theta(log(k!) + (n-k)log(k+1))$ und das lässt sich auf jeden Fall beschreiben als $\Omega(k\ log(k) + (n-k)log(k)) = \Omega(n \cdot log(k))$

3.4 - Zur Party

Wir wollen von unserem Zuhause B zu einer Party A, und dabei auf jeden Fall einen Getränkemarkt besuchen. Die Idee hierbei ist die kürzeste Kombination aus Wegen von B zu einem Markt G_X und von G_X zu A zu finden. Wir können also von Zuhause aus die Länge der kürzesten Wege zu allen Getränkemärkten bestimmen. Zusätzlich lässt sich für jeden Getränkemarkt die kürzeste Verbindung zur Party rausfinden. Dann ist das Minimum aus allen Wegen über die Märkte zur Party der gesuchte Weg.
Algorithmus:
Zunächst führen wir auf dem Graphen eine Breitensuche startend in B aus. In dieser Breitensuche notieren wir zu allen gefundenen Getränkemärkten den Weg dorthin und die jeweilige Länge des Weges. Damit ergeben sich m Paare aus Weg und Länge. Das geschieht in $O(|V| + |E|)$. Um die Wege von den Märkten zur Party zu finden drehen wir alle Kanten um (in $O(|E|)$) und führen eine Breitensuche auf dem inversen Graphen startend in A aus.
Auch hier erhält jeder Markt ein Paar aus Weg und Länge in $O(|V| + |E|)$. Nun gehen wir der Reihe nach jeden Markt durch und vergleichen die Summen der Längen. Wir finden so das Minimum aus allen Doppelpaaren in $O(m)$, $m < |V|$. In diesem Minimum muss nur noch im Weg des zweiten Paares jede Kante zurückgespiegelt werden, das fügt noch maximal $O(|E|)$ hinzu.
Die Laufzeit ist damit $O(|V| + |E| + |V| + |E| + |V| + |E|) = O(|V| + |E|)$
Korrektheit:
Wir müssen mind. einen Getränkemarkt besuchen, daher betrachten wir keinen kürzesten Weg ohne Getränkemarkt. Angenommen es gäbe einen über G_X, der kürzer als unser Minimum wäre, dann wäre der Weg von B zu G_X + der Weg von G_X zur Party kürzer als das Minimum aus allen Summen der beiden Teilwege. dann hätten wir es aber schon gewählt. Der Algorithmus arbeitet also immer korrekt.