Aufgabe 1.1 - Sortieren

Selection Sort:
Gemäß Vorlesung benötigt Selection Sort immer N-1 Iterationen und in jeder i-ten Iteration N-i Vergleiche und höchstens 1 Vertauschung. für a) ergibt sich damit $\Theta(N^2)$ Vergleiche und da in diesem speziellen Fall mindestens N/2 Vertauschungen benötigt werden asymptotisch $\Theta(N)$ Vertauschungen.
Für b) analog $\Theta(N^2)$ Vergleiche und da jeder Wert <N auf jeden Fall getauscht werden muss $\Theta(N)$ Vertauschungen.

Bubble Sort:
Für a): Nach jedem i-ten Durchlauf sind die ersten und letzen i Zahlen sortiert,damit werden immer noch N/2 Durchläufe benötigt und somit $\Theta(N^2)$ Vergleiche und auch $\Theta(N^2)$ Vertauschungen.
Für b): Nach zwei Durchläufen sieht der Zwischenstand wie folgt aus: 2,1,4,3,6,5,...,N-2,N-2,N-1,N, hierfür waren 2N-2 Vergleiche und 2N-3 Vertauschungen nötig. Im nächsten Durchlauf kommen N-1 Vergleiche und (N-2)/2 Vertauschungen hinzu. Danach ist das Ergebnis sortiert. Es folgt der letzte Durchlauf mit N-1 Vergleichen aber 0 Vertauschungen. Also ergeben sich $\Theta(N)$ Vergleiche und auch $\Theta(N)$ Vertauschungen.

Insertion Sort:
Für a):$\Theta(N^2)$ Vergleiche, da jedes (2i+1)-te Element i-Vergleiche hat und dafür auch mindestens i-1 Vertauschungen notwendig sind ergibt sich auch $\Theta(N^2)$ Vertauschungn
Für b) $\Theta(N)$ Vergleiche, da in dieser Folge jede Zahl maximal 4 Vergleiche benötigt. Dies ergibt sich durch die Verteilung von maximal 3 größeren die vorher vorkommen. Auch sind somit maximal 3 Vertauschungen pro Element nötig. Also sind auch die Tausche $\Theta(N)$.

Aufgabe 1.2 - Quicksort

a)
73 , 54 , 63 , 68 , 12 , 75 , 74 , 29 ; mit pivot(links,rechts) = rechts
Wir rufen Quicksort(0,7) auf: Start :
73 , 54 , 63 , 68 , 12 , 75 , 74 , 29 ; pivot = 7 ; A[7]=29

Aufruf partition (7,0,7):
12,54,63,68,73,75,74,29 l = 0, r = 4
12,29,63,68,73,75,74,54 l = 1, r = 0 Ende (tausche pivotelement):
Positionen vor Aufruf: (links = 0, i = 1, rechts = 7)
(12) (29) (63,68,73,75,74,54) !Notation: (LTA)(P)(RTA)!
linkesTA = (12), Pivot =(29), rechtesTA=(63,68,73,75,74,54)
LinkesTA ist sortiert (quicksort(0,0));
Pivot ist sortiert;
12,29 | 63,68,73,75,74,54 → quicksort(2,7) ; A[7] = 54
12,29 | () (54) (68,73,75,74,63) –leeres linkes Teilarray, Pivot=(54)
12,29,54 | 68,73,75,74,63 → quicksort(3,7); A[p] = 63 nach partition:
12,29,54 | () (63) (73,75,74,68) – leeres linkes TA, pivot =(63)
12,29,54,63 | 73,75,74,68 → quicksort(4,7) ; A[p] = 68
12,29,54,63 | () (68) (75,74,73)
12,29,54,63,68 | 75,74,73 → quicksort(5,7) ; A[p] = 73
12,29,54,63,68 | () (73) (74,75)
12,29,54,63,68,73 | 74,75 → quicksort(6,7) ; A[p] = 75
12,29,54,63,68,73 | (74) (75) () – leeres rechtes TA
12,29,54,63,68,73,74,75 sortiertes Array!

b)
73 , 54 , 63 , 68 , 12 , 75 , 74 , 29 ; mit pivot(links,rechts) = links
Wir rufen Quicksort(0,7) auf:
partition auf: 29,54,63,68,12,75,74,73 ,(29 und pivot wurden getauscht)
Nach partition endet l auf 5, r auf 4:
(29,54,63,68,12) (73) (74,75)
rechtes TA wird zwar partitioniert aber nicht sortiert, daher bleibt nur Links:
29,54,63,68,12 | 73 | ()(74)(75) → quicksort(0,4) ; A[0]=29
partition auf : 12,54,63,68,29 ergibt:
(12) (29) (63,68,54) | 73,74,75 → quicksort(2,4) ; A[2]=63
partition auf 54,68,63 ergibt:
12,29 | (54) (63) (68) |73,74,75 Alle rek. Aufrufe terminieren:
12,29,54,63,68,73,74,75 sortiertes Array!

1.3 - Sortieralgorithmen

Wenn immer nur Nachbarn tauschen, muss im worst-case jedes Element $\Theta(n)$ oft tauschen zum sortieren, zum Beispiel bei einer umgekehrt sortierten Liste. Dort muss das erste Element n-1, das zweite n-2, das dritte n-3 bis n/2, und auch für die letzten elemente gilt, das n-te Element benötigt n-1 Tausche , das davor n-2.

Es lässt sich gut erkennen, dass im worst-case $\Theta(n)$ Vertauschungen pro Element nötig sind. Das für n Elemente gibt halt mindestens $\Theta(n \cdot n)$ Vertauschungen gesamt.

Mathematisch können die Fehlstellungen gezählt werden, also wieviele Paare $A[i], A[j], i < j$ es gibt, mit A[i] > A[j]. Für das umgekehrt sortierte Startarray sind das für die Stelle j gerade n-j Fehlstellungen. Das sind $\sum_{j=1}^{n} n-j = \frac{n(n+1)}{2} = \Theta(n^2)$ Fehlstellungen, aber eine einzelne Vertauschung von Nachbarn kann **maximal** eine Fehlstellung beheben. Die Anzahl an Tauschen ist somit größer gleich der Anzahl an Fehlstellungen.
Somit sind $\Omega(n^2)$ Tausche nötig.

1.4 - Schokolade

Auf einer sortierten Liste/Array von Gewichten/Packungen kann für alle Teilfolgen der Länge m geprüft werden, welche Differenz herrscht. Das funktioniert in $\Theta(n)$*, da für sortierte Arrays gilt: Differenz bei Start der Teilfolge in Position n = A[m+n] - A[n]. Hieraus das Minimum ist die gesuchte Menge an Packungen.
*eigentlich sogar in $\Theta(n-m)$, falls n m bedingt.

Können wir Die Packungen noch in $o(n^2)$ sortieren, so ist das Problem gelöst, denn $o(n^2) + \Theta(n) = o(n^2)$. Und aus der Vorlesung wissen wir: Heapsort sortiert in $O(n \cdot log(n))$.

Pseudocode:

```
//Die Packungen liegen im unsortierten Array A
// gegeben sind m und n
minpos = 0

Array B = heapsort(A); // heapsort returned das sortierte Array
// finde minimale Differenz im sortierten Array

mindiff = B[m] - B[0]   //initialisieren

for (i=1 ; (i+m) <= n ; i++){
    if (B[m+i] - B[i]) < mindiff{ //neues minimum bei i
    mindiff = (B[m+i] - B[i]) ; minpos = i;}
    }
return minpos; //ab dieser Packung m abzaehlen.
```

Korrektheit:
Um m Elemente mit minimaler Differenz zu finden, müssen alle in das Intervall [min,max] fallen. In einem sortierten Array liegen alle Elemente zwischen zwei beliebigen Positionen in ebendiesem Intervall. Da die Anzahl vorbestimmt ist, gilt es also die kleinste Differenz einer Teilfolge mit m Elementen innerhalb eines sortierten Arrays zu finden. Genau das erledigt dieser Algorithmus.