Lösningsförslag

Sortering

U 1.
a) Ja. Det blir bara en jämförelse per pass och kostnad $O(n)$.
b) Nej. Metoden gör alltid samma arbete oavsett indata.
c) Ja. Att bygga heapen kostar $O(n)$ om alla är lika. Det är bara att lägga in elementen i vektorn, inga byten behöver ske. Varje `poll` blir $O(1)$ ty inga byten behövs. Totalt $O(n)$.
d) Ja, om man stannar upp och gör byte vid likhet med pivotelementet kommer vektorn varje gång att delas i två lika halvor. Då inträffar bästa fallet för metoden, dvs $O(n * log\ n)$.

U 2. I fall 1 vill man ha en sorteringsmetod som kan avbrytas efter ett antal steg och då ha de k minsta elementen på plats. Då kan man använda urvalssortering. Den finns i två varianter. I vektor där man söker upp minsta element , byter det med elementet på plats 1. Därefter söker man upp det minsta i den delvektor som består av alla element utom det första och byter det till plats 2 o.s.v. Efter k steg har man då de k minsta elementen på rätt plats. Det kostar $O(k * n) = O(n)$ om k är litet. Man kan också använda heapsort och avbryta den efter k steg. Det kostar $O(n + k * \log n) = O(n)$. ($O(n)$ för att bygga heapen och sedan k st `poll`).

I fall 2 passar insättningssortering bra. När det finns få inversioner i vektorn (par av element som ligger fel i förhållande till varandra) så behöver den göra få byten. Insättningssortering blir t.ex. linjär om elementen är sorterade från början.

I fall 3 får vi ta en metod som är snabb i störst allmänhet. Det har visat sig att Quicksort vinner i medeltal även om den har ett dåligt värsta fall.

U 3. `Arrays.sort(v);`
Det är en variant av quicksort som används i just denna `sort`-metod.

U 4.
a) Sortera talen. Detta kostar $O(n \log n)$. Dubbletter hamnar då intill varandra. Gå igenom den sorterade vektorn och tag bort dubbletterna. Detta kostar $O(n)$. Total kostnad: $O(n \log n)$.
Jämför detta med att utföra dubblettborttagning direkt i den osorterade vektorn: För varje element i vektorn behöver vi då söka upp och ta bort dubbletter. Detta blir $O(n^2)$.
b) Sortera båda vektorerna. Det kostar $O(n \log n)$. Sedan gör man som i merge-steget i mergesort. Vid likhet mellan aktuellt element i någon av vektorerna med det element som senast flyttats över i resultatvektorn avstår man dock från att flytta över aktuellt element till resultatet. Sammanslagningen går på linjär tid. Totalt blir det alltså $O(n \log n)$.
Om vektorerna inte är sorterade kan man göra så här: För vart och ett av elementen i den andra vektorn undersöker man om det finns ett likadant element i den första. Om inte lägger man över elementet i resultatvektorn. Om det finns avstår man från att lägga över det. Till sist flyttar man också alla elementen i den första vektorn till resultatvektorn. Algoritmens första del (att leta efter element) är den dyraste. Den kostar $O(n^2)$.
c) Man kan börja med att sortera `m`. För varje element `z` i `m` görs sedan en binärsökning efter elementet `x`−`z` bland de $n - 1$ övriga elementen i `m`. Sorteringen kostar $O(n \log n)$. Det kostar också $O(n \log n)$ att göra de n binärsökningarna. Hela algorit-

men kostar alltså $O(n \log n)$.

Ett annat alternativ är att efter sorteringen av `m` göra så här: Börja med att betrakta det första och det sista talet i `m`, m_1 och m_n. Om $m_1 + m_n = x$ så är vi klara. Om $m_1 + m_n < x$ så vet vi att m_1 inte kan finnas med i lösningen eftersom då $m_1 + m_i < x$ för alla i. Vi kan därför eliminera m_1 och lösa det återstående problemet för $m_2, ..., m_n$. På liknande sätt kan vi eliminera m_n om $m_1 + m_n > x$. Värstafallstiden för denna algoritm blir $O(n)$. Vi eliminerar ju ett av talen i varje steg och varje steg kostar $O(1)$.

U 5.
```
/** Sök upp det i:te elementet i storleksordning i vektorn a. */
public static <T extends Comparable<? super T>> T findIth(T[] a, int i) {
    return findIth(a, 0, a.length - 1,  i - 1);
}

/* Sök upp det i-1:te elementet i storleksordning bland a[first] ... a[last]. */
private static <T extends Comparable<? super T>> T  findIth(T[] a, int first,
                                                            int last, int i) {
    int pivIndex = partition(a, first, last);
    if (i == pivIndex) {
        return a[i];
    } else if (i < pivIndex) {
        return findIth(a, first, pivIndex - 1, i);
    } else {
        return findIth(a, pivIndex + 1, last, i);
    }
}
```