Svar och kommentarer till tentamen 2018-06-08 i DA3018

1. I ett binärt sökträd gäller att, för varje hörn v med vänsterbarn u och högerbarn w, så är nycklarna i delträdet T_u mindre än nyckeln för v, och nycklarna i T_w är större än nyckeln för v. Man ska kunna sätta in nya element, ta bort hörn med en viss nyckel, samt hitta hörn som har en viss nyckel.

2. Sondering är när man lagrar elementen och deras nycklar direkt i hashtabellen, och alltså inte använder länkade listor för att hantera krockar, utan letar efter tomma platser på alternativa (tomma) platser i tabellen.

3. Bubbelsortering

   (a) Koden har två nästlade loopar. Innerst sker en jämförelse och ibland ett anrop till `swap`, som vi kan anta tar $O(1)$ tid. Den inre loopen gör j iterationer, och den yttre gör $n = |data|$ iterationer. Det ger totalt $\sum_{j=0}^{n-1} \sum_{i=0}^{j-1} O(1) = \sum_{j=0}^{n-1} O(j) = O(n^2)$.

   (b) Ja, det är i befintligt minne eftersom ingen ytterligare datastruktur görs. Koden ändrar ordningen på elementen i indata-vektorn.

   (c) Ja. Algoritmen byter bara plats på angränsande element, och endast då det ena är strikt större än det andra.

   (d) Anpassningen nedan håller koll på om omkastningar har gjorts. `n_swaps == 0` så har alltså inga element bytt plats i den inre loopen, så elementen på plats 0 till `iter_numer` är i rätt ordning. Tidigare iterationer i den yttre loopen har garanterat att elementen från `j` och uppåt är i rätt ordning. Alltså kan man sluta sortera.

   Tidskomplexiteten påverkas inte, eftersom det är en värsta-fall-analys man bör göra, och det kan fortfarande krävas $O(n^2)$ omkastningar.

```
def bubble_sort(data):
    for j in range(len(data)-1, 0, -1):  #start, stop, and
         step
        n_swaps = 0
        for i in range(j):
            if data[i] > data[i+1]:
                swap(data, i, i+1)
                n_swaps += 1
        if n_swaps == 0:
            return data
    return data
```

4. Både medel och median går att beräkna i linjär tid. Men vi kan göra det enkelt för oss genom att sortera indata, exv med hjälp av mergesort. Det tar tid $O(n \log n)$ vilket normalt beräknas som effektivt. Här är en enkel implementation:

   (a)
```
def medel_median(data):
    s_data = sorted(data)     # nlogn
    l = len(data)             # n
    median = s_data[l//2]     # 1
    medel = sum(data)/l       # n
```

```
n = 0                       # 1
small = min(median, medel)  # 1
large = max(median, medel)  # 1
for e in data:              # n iterationer
    if small < e and e < large:   # 1
        n += 1                    # 1
return n
```

(b) Kommentarerna i koden anger tidskomplexiteten för de enskilda raderna. Sammantaget är det $O(n \log n) + O(n) + O(1) + O(n) + O(1) + O(1) + O(1) + O(n) = O(n \log n)$.

5. Alg1 har tidskomplexitet $O(n^2)$, så vi kan anta att körtiden approximeras av Cn^2 för någon konstant C som är antalet operationer som behövs. En körning av Alg1 på 10^6 element ger alltså att tiden per operation är $10/(C(10^6)^2) = 10^{-11}/C$. Alg2 har komplexitet $O(n \log_2 n)$, så dess körtid är ungefär $Kn \log_2 n$. För $n = 10^6$ blir det ca $10^6 \cdot 6 \cdot \log_2(10)K \approx 2 \cdot 10^7 K$.

   Körtiden för Alg2 kan då skattas till $2 \cdot 10^7 K \cdot 10^{-11}/C = 2K/C \cdot 10^{-4}$. Även om K är en storleksordning större än C kommer vi alltså ner på millisekunder istället för 10s.

   Det går säkert att dra denna slutsats både snyggare och snabbare!

6. (a) För ett träd T löser följande pseudokod problemet med ett anrop till `list_nodes_inorder(root(T))`.

```
def list_nodes_inorder(v):
    if v:
        left_nodes = list_nodes_inorder(v.left)
        right_nodes = list_nodes_inorder(v.right)
        return left_nodes ++ [v] ++ right_nodes
    else:
        return []
```

   (b) Vi kan argumentera induktivt. Om T är tom returneras tomma listan, vilket är korrekt. Om T har endast ett hörn returneras `[] ++ [v] ++ []`, vilket är korrekt. Antag nu att koden är korrekt för träd mindre än något T. Det kommer då att göras två rekursiva anrop, ett för vänster och ett för höger delträd. Listan för vänster-anropet kommer för v, som kommer före högeranropet, och därmed returneras hörnen i *inorder*.

   (c) Algoritmen kommer att göra ett rekursivt anrop för varje hörn i trädet. Arbetet i varje rekursivt anrop är $O(1)$. Därmed blir den tidskomplexiteten $O(|T|)$, dvs linjär.

7. (a) Det räcker med $O(n)$ tid att bygga en heap.

   (b) En enkel max-heap är $h = (7, 6, 5, 4, 3, 2, 1)$: här är h_i alltid större än h_{2i} och h_{2i+1}.

   (c) Den etablerade algoritmen för att utöka en heap lägger det nya elementet sist, och utför därefter iterativt operationen *heapify* på dess förälder. Det skulle stegvis ge en array som ser ut så här:

   - $h = (7, 6, 5, 4, 3, 2, 1, 8)$ — vi har lagt elementet sist.
   - $h = (7, 6, 5, 8, 3, 2, 1, 4)$ — efter `heapify(h,4)`, element h_4 är förälder till element h_8.

- $h = (7, 8, 5, 6, 3, 2, 1, 4)$ — efter `heapify(h,2)`
- $h = (8, 7, 5, 6, 3, 2, 1, 4)$ — efter `heapify(h,1)`

8. (a) Indata: en mängd mätningar $M = \{v : v \in R^n\}$ och ett tröskelvärde δ som begränsar hur nära mätningarna får vara varandra.
   Utdata: största möjliga delmängd $M' \subseteq M$ så att $\forall u, v \in M' : |u, v|_2 \leq \delta$.
   Det här problemet är en variant av "oberoende mängd".

   (b) En snål algoritm är efterfrågad eftersom oberoende mängd är ett beräkningsmässigt svårt problem.

```
maximal_independent(M, delta):
    Msub = {}
    for v in M:
        n = 0
        for u in Msub:
            if distance(u, v) > delta:
                n = n + 1
        if n == 0:
            Msub = Msub + {v}
    return Msub

distance(u, v):
    d = 0
    for i = 0 to n-1:
        d += abs(u[i] - v[i]) ** 2
    return sqrt(d)
```

   Denna algoritm returnerar en *maximal* oberoende mängd, dvs vi kan inte lägga till ett till element från M utan att bryta mot villkoret.

   (c) För tidskomplexiteten noterar vi att funktionen `distance` gör n iterationer med konstant arbete, så $O(n)$.
   `maximal_independent` itererar över varje element i M, och i varje iteration testas om det finns ett element i M' som är för nära. I värsta fall har vi efter i iterationer lagt till $i - 1$ element till M' och därför görs $\sum_{i=1}^{|M|} i$ jämförelser, utökningar av M', samt anrop till `distance`. De två första operationerna tar $O(1)$ tid. Sammantaget tar därför heuristiken $O(\sum_{i=1}^{|M|} in) = O(m^2 n)$ tid.

9. (a) Man kan anpassa djupet-först-sökning för detta problem:

```
def get_tree(V, E):
    v = V[0] # Pick a start vertex
    return = visit(v)

def visit(v):
    v.visited = True
    E_T = {} # The empty set
    for u in neighbor(v):
        if not u.visited:
            E_T = union(E_T, {(v, u)})
            E_T = union(E_T, visit(u))
    return E_T
```

   (b) Djupet-först-sökningen traverserar hela grafen och lägger till en kant (v, u) närhelst vi går till ett hörn u som inte tidigare besökts. Det kan därför inte bildas cykler i E_T, så det är alltid ett träd som returneras.

(c) Vi väljer ett starthörn i tid $O(1)$. Om mängden E_T representeras med en lista kan vi lägga till en kant i tid $O(1)$. Om listan är representerad så att vi kommer åt första och sista elementet kan vi också beräkna beräkna union i konstant tid. Det rekursiva anropet sker för hörn vi inte besökt, så det blir $O(|V|)$ anrop till `visit`. Loopen är över grannarna till v och ger därför $O(\text{grad}(v))$ varv, men sammanlagt sker $O(|E_G|)$ iterationer, och algoritmens tidskomplexitet blir därför $O(|V| + |E|)$.