Homework problems:

4. [Dasgupta et al., Ex. 6.1] A contiguous subsequence of a list $S$ is a subsequence made up of consecutive elements of $S$. For example, if $S$ is

$$5, 15, -30, 10, -5, 40, 10$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

**Input:** A list of numbers, $a_1, a_2, \ldots, a_n$.

**Output:** A contiguous subsequence of $a_1, a_2, \ldots, a_n$ that has the maximum possible sum (the sum of an empty subsequence is zero).

For the preceding example, the output is $10, -5, 40, 10$, with a sum of $55$.

**Hint:** For each $1 \leq j \leq n$, consider contiguous subsequences that end at position $j$. Note that every contiguous subsequence that ends at $a_j$ (other than the contiguous subsequence $(a_j)$) is an extension of a contiguous subsequence that ends at $a_{j-1}$.

5. [Dasgupta et al., Ex. 6.22] Give an algorithm with running time $O(nt)$ for the following task.

**Input:** A list of $n$ positive integers $a_1, a_2, \ldots, a_n$ and a positive integer $t$.

**Question:** Decide (output "yes" or "no") whether there is a subset of the $a_i$'s whose sum is equal to $t$. Each $a_i$ may be used at most once.

**Hint:** Let $T(v, j)$ indicate whether there exists a subset of the first $j$ numbers which sums to $v$.

6. Prove the following basic facts about polynomial-time reducibility and complexity classes:

   (i) If $S \leq^P T$ and $T \leq^P U$, then $S \leq^P U$.
   (ii) If $S \leq^P T$ and $T \in P$, then $S \in P$.
   (iii) Let $T$ be an NP-complete problem. If $T \in P$, then $P = NP$.
   (iv) Let $S$ be some NP-complete problem, $T \in NP$ and $S \leq^P T$. Then also $T$ is NP-complete.

**Hint:** Understand the definitions of the complexity classes and the notion of polynomial-time reduction. Utilize function compositions and note how polynomial functions compose.

Demonstration problems:

7. [Dasgupta et al., Ex. 6.21] A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in $E$. Give a linear-time algorithm for the following task:

   **Input:** A tree $T = (V, E)$.
   **Output:** The size of a smallest vertex cover of $T$. 
For instance, in the following tree, the possible vertices include \( \{A, B, C, D, E, F, G\} \) and \( \{A, C, D, F\} \) but not \( \{C, E, F\} \). The size of a smallest vertex cover is 3. Indeed, \( \{B, E\} \) is such a vertex cover, and no set of two vertices is a vertex cover.

(Hint: Have a look at the dynamic programming algorithm for independent sets on trees presented in Lecture 9.)

**Solution:** This solution is similar to the 'Independent sets in trees' algorithm in Lecture 9.

One dfs or bfs run in \( O(|V|) \) can arrange the tree as a rooted tree such that all nodes have a unique parent (except the root). Each node \( v \in V \) now defines a unique subtree \( T(v) \) hanging from it (consider removing the edge between the node and its parent). In the example tree, if \( B \) is the root then \( T(E) \) contains nodes \( \{E, D, F, G\} \).

Let \( S(v) \) be the size of a smallest vertex cover in the subtree \( T(v) \), and refer to an example of such cover by \( \gamma(v) \). Note that the minimum vertex cover of the whole tree corresponds to \( S(v_{\text{root}}) \). Thus, we aim to find \( S(v) \) by dynamic programming for all \( v \) and return \( S(v_{\text{root}}) \) as the final result.

For each \( v \in V \), a minimum vertex cover \( \gamma(v) \) for the subtree \( T(v) \) either contains \( v \) or it doesn’t contain \( v \). In the first case with \( v \in \gamma(v) \), one may consider erasing the edges adjacent to \( v \) as already taken care of, and one needs to look only at the forest of subtrees \( T(c) \) for \( c \in C(v) \), where \( C(v) \) denotes the set of children of \( v \) in the rooted tree. Now the vertex cover \( \gamma(v) \) must also contain (as subsets) some minimum vertex covers \( \gamma(c) \) for the children’s subtrees.

In the latter case with \( v \notin \gamma(v) \), the edges between \( v \) and each \( c \in C(v) \) force each child to be in the vertex cover \( \gamma(v) \). Erasing the now-covered edges, the only remaining part is the forest of grandchildren’s \( gc \in GC(v) \) subtrees \( T(gc) \). Then \( \gamma(v) \) also contains minimal vertex covers \( S(gc) \) for each \( gc \in GC(v) \).

Picking the better of these two cases leads to the recursion:

\[
S(v) = \min \{ 1 + \sum_{c \in C(v)} S(c), \quad |C(v)| + \sum_{gc \in GC(v)} S(gc) \}.
\]

Note that for leaves \( v_l \) this formula gives directly the correct value \( S(v_l) = 0 \).

A recursive implementation is given as a pseudocode. The initialization goes through each vertex in the tree once and thus is linear in \( |V| \). The evaluate-size recursion call tree is the same as the input tree traversed in a depth-first fashion and thus there are \( O(|V|) \) recursive calls. All the other computations in evaluate-size are constant time with respect to \( |V| \) and thus the whole algorithm takes \( O(|V|) \) time.
Algorithm 1: Algorithm for minimum vertex cover in trees

1 function min-vertexcover(v);
   Input: Undirected tree \( T = (V, E) \)
   Output: Size of minimum vertex cover in \( T \)
2 Select a root \( v_0 \in V \);
3 Use dfs from \( v_0 \) to find a parent for each \( v \in V \setminus \{v_0\} \);
4 \( S \leftarrow \) empty table with index set \( V \);
5 \( CS \leftarrow \) empty table with index set \( V \) for maintaining the sum of
   minimum vertex covers of the children of the vertex;
6 for each \( v \in V \) do
7   \( S[v] \leftarrow \) unknown;
8 end
9 evaluate-size(\( v_0 \), \( S \));
10 return \( S[v_0] \);
11 ;
12 function evaluate-size(\( v \), \( S \));
13 value-self \( \leftarrow \) 1;
14 value-noself \( \leftarrow \) 0;
15 \( CS[v] \leftarrow 0; \)
16 for \( c \in C(v) \) do
17   if \( S[c] \) unknown then
18      evaluate-size(\( c \), \( S \));
19      value-self \( \leftarrow \) value-self + \( S[c] \);
20      value-noself \( \leftarrow \) value-noself + \( CS[c] + 1; \)
21      \( CS[v] \leftarrow CS[v] + S[c]; \)
22   end
23 end
24 \( S[v] \leftarrow \min \{ \text{value-self, value-noself} \}; \)
25 return ;