NP-completeness I

Readings: Cormen et al, page 99 in the readings.

Recommended: *Computers and Intractability*, Garey and Johnson, chapters 1-3.

Search problems: For each instance I of a search problem there is a search space $S(I)$, which can be large. We want to find some element $S \in S(I)$ satisfying a condition $C(S)$, or find $M = \max\{v(S) | S \in S(I) \text{ and } C(S)\}$ or $m = \min\{v(S) | S \in S(I) \text{ and } C(S)\}$.

SKS:

Instance: $I = \langle d, w_1, \ldots, w_d, C \rangle$, $w_i \in \mathbb{N}$, $S(I) = \{S | S \subseteq \{1, 2, \ldots, d\}\}$

Problem: Find $M = \max\{K(S) | K(S) \leq C\}$

SKS_2:

Instance: Same as SKS, with the same search space $S(I)$.

Problem: Find $S_0 \subseteq \{1, 2, \ldots, d\}$ s.t. $K(S_0) = M(I)$.

Many optimization problems come in two versions:

1. Find an optimal value.
2. Find an actual structure that gives that value.

Reductions: Reduction is a fundamental notion in NP-completeness. The idea is the following: problem P_1 is reducible to problem P_2 if P_1 can be solved with the help of a P_2 solver.

Definition 1 P_1 is polytime reducible to P_2 (denoted $P_1 \xrightarrow{p} P_2$) if there is a polytime algorithm for P_1 which is allowed to access a solver for P_2. The time taken by the P_2 solver is not counted.

Example 1 JSP is the Job Selection with Profits problem. To show that SKS $\xrightarrow{p}$ JSP, we solve SKS using JSP. Namely, given an instance $\langle d, w_i, \ldots, w_d, C \rangle$ of SKS, devise an instance of JSP with the same solution as follows: $n = d$, $d_i = w_i$, $t_i = C, p_i = w_i$. Now, if we can solve JSP, then we can solve SKS.

[1]These notes were taken originally by Antonina Kolokolova in Steven Cook's csc364 lecture, Spring 99, and they have been edited for this term.

It is important to understand the direction of reductions. In general:

$$\text{special case} \xrightarrow{p} \text{general case} \qquad \text{and} \qquad \text{easy} \xrightarrow{p} \text{hard}$$

Decision problems: A decision problem assigns either "YES" or "NO" to each instance. In general, we can associate a decision problem with a search problem. We can reduce a search problem to its associated decision problem and vice versa, so they are "equally hard".

SKD - decision problem for SKS:

Instance: $I = \langle d, w_1, \dots, w_d, C, B \rangle$, where $B \in \mathbb{N}$ is an extra parameter denoting the lower bound for $K(S)$.

Question: Is there a subset of indices such that $B \leq K(S) \leq C$?

Claim 1 SKS $\xrightarrow{p}$ SKD

Proof of Claim. Fix an instance $I = \langle d, w_1, \dots, w_d, C \rangle$ of SKS. We want to find $M(I)$ where $M(I) = \max\{K(S) | K(S) \leq C\}$.

Let SKD(I, B) be the answer to SKD for the instance I, B.

Idea: try $B = C$. If SKD(I, C)="YES", then $M(I) = C$. If the answer is "NO", then we can use binary search: call SKD$(I, \lceil \frac{C}{2} \rceil)$. If the answer is "YES", call SKD$(I, \lceil \frac{3C}{4} \rceil)$. If the answer is "NO", call SKD$(I, \lceil \frac{C}{4} \rceil)$. Continue reducing the size of the interval by 2 in each step, until a value B is obtained such that SKD(B) ="YES" and SKD$(B+1)$ ="NO". That B will be the solution to SKS.

The main reason to use binary search instead of linear search (trying for $B = C-1$, $B = C-2$ and so on) is that binary search is $O(\log C)$, whereas linear search is $O(C)$ and thus is not polytime when C is written in binary.

SKD$_2$ decision problem for SKS$_2$:

Instance: $\langle d, w_1, \dots, w_d, C, S \rangle$, $S \subseteq \{1, 2, \dots, d\}$.

Question: Can S be extended to an optimal solution S_0 of SKS$_2$ i.e. is there a set $S_0 \subseteq \{1, 2, \dots, d\}$ s.t. $S \subseteq S_0$ and $K(S_0) = M(I)$?

Exercise: Show SKS$_2 \xrightarrow{p}$ SKD$_2$.

Languages: We translate each decision problem into a set of strings. Σ is a finite alphabet of symbols, e.g. $\Sigma = \{0, 1, \#\}$, and Σ^* is a set of finite strings over Σ. Code I by $e(I) \in \Sigma^*$. If $m \in \mathbb{N}$, $\bar{m} \in \{0,1\}^*$ is m in binary (e.g. $\bar{6} = 110$).

For $I = \langle d, w_1, \dots, w_d, C \rangle$, one possible encoding is $e(I) = \bar{d}\#\bar{w}_1\#\dots\#\bar{w}_d\#\bar{C}$, $e(I) \in \{0, 1, \#\}^*$. For example $e(\langle 2, 5, 6, 3 \rangle) = 10\#101\#110\#11$.

Definition 2 A language L over Σ is a subset of Σ^*. The language L_{P}, associated with a decision problem P is a set of strings $L_{\mathrm{P}} = \{x \in \Sigma^* | x \text{ codes a "YES" instance of P}\}$

in Cormen: L is "concrete decision problem".

Example 2

$$L_{\mathrm{SKD}} = \{e(\langle d, w_1, \ldots, w_d, C, B\rangle) | \langle d, w_1, \ldots, w_d, C, B\rangle \text{is a "YES" instance to SKD}\}$$

So, decision problems turn into languages.

Definition 3 A language $L \subseteq \Sigma^*$ is a *polytime language* iff there is a polytime algorithm which on input $x \in \Sigma^*$ determines whether $x \in L$. We let P denote the class of polytime languages.

Conjecture: $L_{\mathrm{SKD}} \notin P$

Note that the above definition of polytime languages is informal since we don't have a precise definition of algorithms. Thus, we need to introduce Turing Machines to formalize the notion of algorithm.

Turing machines

Ref: Readings: Sipser Chap 3, page 147 in the readings.

Turing machines were proposed by Alan Turing in 1936.

A *Turing machine* consists of a two-way infinite tape and a finite state control. The tape is divided up into squares, each of which holds a symbol from a finite tape alphabet (usually $\{0, 1, b\}$, where b corresponds to blank).

A Turing machine M has a read/write head, that scans squares on the tape. Depending on the state of the control and the symbol scanned, it makes a move, consisting of the following: printing a symbol, shifting the head left or right one square, and assuming a new state.

A Turing machine M is specified as follows:

1. Γ (finite tape alphabet), Σ (finite input alphabet), and $\Gamma = \Sigma \cup \{b\}$ where (b is the "blank" symbol).
2. Q (finite set of states). There are 3 special states: q_0 (initial state), q_{accept} (state in which M halts and accepts), and q_{reject} (state in which M halts and rejects).
3. $\delta : (Q - \{q_{accept}, q_{reject}\} \times \Gamma) \to Q \times \Gamma \times \{L, R\}$

$\delta(q, s) = (q', s', h)$, where q is the current state, s is the scanned symbol, q' is the new state, s' is the printed symbol and h is either L or R, corresponding to the left or right shift.

Input conventions: the input string $x \in \Sigma^*$ is written on the tape surrounded by blanks, and M starts in state q_0 scanning the leftmost symbol of x, or b if x is the empty string.

Example 3 We want to design a TM which adds 1 (in binary). Let $\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$ and $Q = \{q_0, q_{\text{accept}}, q_{\text{reject}}, q_1\}$. The δ is specified by the following table:

State	Symbol	Action
q_0	0	q_0, print 0, R
q_0	1	q_0, print 1, R
q_0	b	q_1, print b, L
q_1	0	q_{accept}, print 1, L
q_1	1	q_1, print 0, L
q_1	b	q_{accept}, print 1, L

Definition 4 A TM M *accepts* a string x if, when we run M on input x, it eventually halts in state q_{accept}. A language accepted by M is $L(M) = \{x \in \Sigma^* | M \text{ accepts } x\}$.

Exercise: PAL = even length palindromes = $\{yy^r | y \in \{0, 1\}^*\}$, where y^r means y spelled backwards. Design a TM such that $L(M) = \text{PAL}$.