Efficient Algorithms

Previously, we rigorously defined what is meant by an algorithm. Now, we can apply the same ideas and define was it means for an algorithm to be efficient. Formal definitions are not usually required when we design algorithms for problems as long as we are coming up with descriptions of procedures that people accept as sufficiently efficient algorithms. However, just as we required formal definitions in order to argue that algorithms for a particular problem do not exist, we also must be completely rigorous about our definitions if we want to start arguing that efficient algorithms (for a particular problem) don't exist.

For formally defining what we mean by an efficient algorithm, we will return to the Turing machine we previously studied. Although Turing was only interested in defining the notion of "algorithm", his model is also good for defining the notion of "polynomial-time" algorithm. (We tend to use "polynomial time" as a rigorous surrogate for "efficient"; this is very useful, even though not all polynomial-time algorithms are truly efficient.)

Recall our definition of the worst case time complexity of Turing machines. Let M be a Turing machine over input alphabet Σ. For each $x \in \Sigma^*$, let $t_M(x)$ be the number of steps required by M to halt on input x.

Definition 1 (Worst case time complexity of M) *The worst case time complexity of M is the function $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined by*

$$T_M(n) = \max\{t_M(x) \mid x \in \Sigma^*, |x| = n\}.$$

Now we are ready to formally define the class $\mathbf{P}$ of polynomial-time languages.

Definition 2 (Polynomial-time Turing Machine) *A Turing Machine M is* polynomial-time *if for some $c \in \mathbb{N}$, $T_M(n) \in O(n^c)$.*

Definition 3 (Formal definition of P)

$$\mathbf{P} = \{L \mid L = \mathcal{L}(M) \text{ for some polynomial-time deterministic Turing Machine } M\ \}.$$

For example, PAL$\in\mathbf{P}$. (Please refer to the definition of PAL in the notes on Turing machines.) We will discuss other languages in $\mathbf{P}$ below.

Before we do, recall that we can view a Turing machine as not only accepting a language, but also as computing a function by outputting a string.

Specifically, we say that M computes $f : \Sigma^* \to \Sigma^*$ if for every $x \in \Sigma^*$, M computing on input x eventually halts with output $f(x)$.

We can also formally define the set $\mathbf{FP}$ of functions computable in polynomial time.

Definition 4 (Formal definition of FP)

$$\mathbf{FP} = \{f : \Sigma^* \to \Sigma^* \mid M \textit{ computes } f \text{ for some polynomial-time Turing machine } M\ \}.$$

Before giving some more examples of languages in $\mathbf{P}$ and functions in $\mathbf{FP}$, we should reflect on the reasonableness of our definitions of these classes. Recall that, except for nondeterminism, the various improvements we made to the Turing machine (such as the multi-tape Turing machine) only resulted in a polynomial speed-up. Thus, their polynomial-time power is just the same as normal Turing machines.

The Church-Turing Thesis, which was previously discussed, states that Turing machines properly formalize the notion of what we can compute. Now, we will state a version of this thesis that states that we have properly formulated the notion of polynomial-time computation.

Polytime Thesis If a language can be accepted (or a function computed) by some sort of polynomial-time algorithm, according to some reasonable notion of polynomial-time algorithm, then that language or function can be computed by a polynomial-time Turing machine.

Although it is intuitively obvious that every polynomial-time Turing machine should be considered a polynomial-time algorithm, it is not at all clear that the converse of this, namely the Polytime Thesis, is true. There are a lot of objections one can think of. An obvious one is that maybe polynomial-time Random Access Machines (or RAMs) are more powerful than Turing machines. However, it is not hard to see how to simulate a polynomial-time RAM by a polynomial-time Turing machine, assuming some reasonable restrictions are placed on our definition of the RAM. (In particular, we insist that the word size is no more than a polynomial in the length of the input.)

A more potent objection is that it often appears to help if we are allowed to toss random coins during a computation; in this case, it is only required that we get the right answer with high probability. This objection can be countered by allowing our Turing machine to flip random coins as well. Another objection involves "quantum computers". We will not describe these here, except to say that there are certain tasks, such as factoring large integers (presented in binary), that quantum computers can perform in polynomial time, that *appear* to require exponential time on even probabilistic Turing machines. It is not clear, however, that it is possible to "physically construct" actual machines that approximate the behavior of quantum computers. It turns out that understanding what problems require exponential time is a very deep mathematical issue; it is also an extraordinarily deep issue in physics to understand to what extent a quantum computer can be built. The point of the Polytime Thesis, is that if we are able to construct machines that are much more powerful than Turing machines (at least allowing randomness), then our understanding of mathematics and of the universe will have been significantly altered.

One should note that that every language in $\mathbf{P}$ has a corresponding function in $\mathbf{FP}$. Let $L \subseteq \Sigma^*$ be a language, where Σ contains both 0 and 1. Define the *characteristic function of*

L, $C_L : \Sigma^* \to \Sigma^*$, by
$C_L(x) = 1$ if $x \in L$, and $C_L(x) = 0$ if $x \notin L$.
It is now easy to see that
$L \in\mathbf{P} \Leftrightarrow C_L \in\mathbf{FP}$.

Note that the definitions of the classes **P** and **FP** are sensitive to how the input is represented. For example, if the input numbers to the knapsack problem are expressed in unary notation, then the dynamic programming problem discussed earlier in the course solves this problem in polynomial time. However, if the numbers are expressed in binary notation (which is the default case), then this problem will turn out to be "**NP**-complete", which will imply that it probably cannot be solved in polynomial time.

Examples of Languages and Search Problems

Often we will describe a language in the following manner.

MSTD (Minimum Spanning Tree Decision Problem).
Instance:
$\langle G, B\rangle$ such that G is a connected, undirected graph with integer costs on the edges, and $B \in \mathbb{N}$ (with all integers represented in binary).

Acceptance Condition:
Accept if G has a spanning tree with cost $\leq B$.

The notation $\langle G, B\rangle$ means that the graph G and number B are represented in some standard form over a standard alphabet. By "Instance", we mean any string that is properly formed, in the manner described. Formally speaking, MSTD is just the set of $\langle G, B\rangle$ such that G is a connected, undirected graph with integer costs on the edges, and $B \in \mathbb{N}$, and G has a spanning tree with cost $\leq B$.

Claim MSTD $\in$ **P**.
A polynomial-time algorithm is: use Kruskal's algorithm to find a MST T; if T has cost $\leq B$, accept, otherwise reject.

The decision problem MSTD is related to the more natural "search problem" MST.

MST (Minimum Spanning Tree Search Problem).
Instance:
$\langle G\rangle$ such that G is a connected, undirected graph with integer costs on the edges (with all integers represented in binary).

Output:
Output a minimal cost spanning tree of G.

We will often describe search problems in this way. The goal, given an instance, is to find some appropriate output such as an optimal value or an optimal structure of some sort; often there are many different possible correct outputs.

We will say that a *search problem is computable in polynomial time*
if for some function $f \in \mathbf{FP}$, $f(x)$ is a correct output for every instance x,
and $f(x)$ is the empty string for all non-instances x.
(The way we have chosen to deal with non-instances is very arbitrary. We could have instead used the convention that we regard non-instances as a particular, trivial instance.)

If we can solve a search problem in polynomial time, then we can solve the corresponding decision problem in polynomial time. Often the converse holds as well.

GKS (General Knapsack Search Problem).
Instance:
$\langle (w_1, g_1), \cdots, (w_m, g_m), W \rangle$ (with all integers represented in binary).

Output:
Output a feasible knapsack with highest possible profit.

GKD (General Knapsack Decision Problem).
Instance:
$\langle (w_1, g_1), \cdots, (w_m, g_m), W, B \rangle$ (with all integers represented in binary).

Acceptance Condition:
Accept if there is a feasible knapsack with profit $\geq B$.

SDPDS (Scheduling with Deadlines, Profits and Durations Search Problem).
Instance:
$\langle (d_1, g_1, t_1), \cdots, (d_m, g_m, t_m) \rangle$ (with all integers represented in binary).

Output:
Output a feasible schedule with highest possible profit.
(Note that the ordering of deadlines, profits and durations is different than in earlier notes: the $\{d_i\}$ are deadlines, the $\{g_i\}$ are profits, and the $\{t_i\}$ are durations.)

SDPDD (Scheduling with Deadlines, Profits and Durations Decision Problem).
Instance:
$\langle (d_1, g_1, t_1), \cdots, (d_m, g_m, t_m), B \rangle$ (with all integers represented in binary).

Acceptance Condition:
Accept if there is a feasible schedule with profit $\geq B$.

We will see in the next section that there are close relationships between these last four problems. Later on, we will see that because the languages are "**NP**-complete", there is very good reason to believe that none of these four problems have polynomial time algorithms.

Polynomial-Time Reducibilities

Let P_1 and P_2 be "problems", where by a "problem" we mean a either a language or a search problem. We will say that P_1 is polynomial-time (Turing) reducible to P_2, and write $P_1 \xrightarrow{p} P_2$, if P_1 can be solved in polynomial time with the help of a P_2 solver.

Definition 5 *P_1 is polynomial-time (Turing) reducible to P_2 ($P_1 \xrightarrow{p} P_2$) if there is a polynomial-time algorithm for P_1 which is allowed to access a solver for P_2, where the time taken by the P_2 solver is not counted.*

It will always be the case that the decision version of a problem is polynomial-time (Turing) reducible to the search version.

Example 1 (GKD $\xrightarrow{p}$ GKS, and SDPDD $\xrightarrow{p}$ SDPDS)

To show, say, the first reduction, we want to solve GKD using GKS. Given an instance $x = \langle (w_1, g_1), \cdots, (w_m, g_m), W, B\rangle$ of GKD, just create $y = \langle (w_1, g_1), \cdots, (w_m, g_m), W\rangle$ and give y to a solver for GKS, getting S as an output. Next, we compute the profit g of S, and compare it to B. If $B \leq g$ then we accept, otherwise we reject. Clearly this only takes time polynomial in $|x|$ (ignoring the time taken by the GKS solver).

Intuitively it is clear that any problem that is polynomial-time (Turing) reducible to a problem solvable in polynomial time, is itself solvable in polynomial time.

Theorem 1 *If $P_1 \xrightarrow{p} P_2$ and P_2 is solvable in polynomial time, then P_1 is solvable in polynomial time.*

Proof:
Say that M_2 solves P_2 in polynomial time. Let M be a Turing machine that solves P_1 in polynomial time, using a solver for P_2. (At this point, we should really give more of an explanation about our syntactic conventions whereby M gets access to a solver for P_2. One way to do this is to have a special tape where M writes inputs for P_2, and another special tape where the output from P_2 instantly appears.) Since M runs in time, say, $O(n^c)$ on inputs of length n, all of the inputs to the P_2 solver must be of length $O(n^c)$.

Assume that M_2 solves P_2 in time $O(n^d)$. We now describe a machine M_1 that solves P_1 in polynomial time. On an input of length n, M_1 will behave like M, but whenever M wants to solve a P_2 problem on an input x, M_1 will run M_2 on x; since $|x| \in O(n^c)$, this running of M_2 will take time $O(|x|^d)$, that is, time $O(n^{cd})$. M_1 will have to run M_2 at most $O(n^c)$ times, for a total running time of $O(n^{c+cd})$. $\square$

It will often be the case that the search version of a problem is polynomial-time (Turing) reducible to the decision version.

Example 2 (GKS $\xrightarrow{p}$ GKD, and SDPDS $\xrightarrow{p}$ SDPDD)

We will only prove the first reduction; the second is similar and is left as an exercise.

Say that we are given an instance $x = \langle (w_1, g_1), \cdots, (w_m, g_m), W \rangle$ of GKS, where $|x| = n$. Our first goal is to find the value of the optimal profit, using a GKD solver. We are able to test, for any B we choose, whether or not it is possible to achieve profit at least B. Let GKD(B) be answer from GKD for the instance $x = \langle (w_1, g_1), \cdots, (w_m, g_m), W, B \rangle$. We *could* perform this test for $B = 1, B = 2, \cdots$ until we get the answer "no". However, since the profits $\{g_i\}$ are written in binary notation, the optimal profit can be as big as (about) 2^n, so this wouldn't run within polynomial time. Instead we will do what is in effect a binary search. Let C be the sum of all the g_i.

We will call GKD($\lceil \frac{C}{2} \rceil$).
If the answer is "yes", call GKD($\lceil \frac{3C}{4} \rceil$); if the answer is "no", call GKD($\lceil \frac{C}{4} \rceil$).
We continue reducing the size of the interval by a factor of 2 on each step, until a value B is obtained such that GKD(B) = "yes" and GKD($B+1$)="no". That B will be the value of the optimal profit.

Now, knowing the optimal profit B, we want to find a knapsack achieving profit B.
If GKD on $\langle (w_1, g_1), \cdots, (w_{m-1}, g_{m-1}), W, B \rangle$ returns "yes", then we can forget about item m, and find a solution to the $\langle (w_1, g_1), \cdots, (w_{m-1}, g_{m-1}), W \rangle$ knapsack problem with (optimal) profit B;
otherwise, we use item m, and find a solution to the $\langle (w_1, g_1), \cdots, (w_{m-1}, g_{m-1}), W - w_m \rangle$ knapsack problem with (optimal) profit $B - g_m$.
Continuing in this way, we find an optimal knapsack. It is easy to check that the running time is polynomial in n. $\square$

An important special case of $L_1 \xrightarrow{p} L_2$ is where L_1 and L_2 are languages, and the reduction is of a very special form: given input x, compute $f(x)$, and accept if and only if $f(x)$ is in L_2. We write $L_1 \leq_p L_2$, and say that L_1 is *polynomial-time (ManyOne) reducible*, or also *polynomial-time transformable*, to L_2.

Definition 6 *Let $L_1, L_2 \subseteq \Sigma^*$.*
Then $L_1 \leq_p L_2$ if for some $f : \Sigma^* \to \Sigma^*$,
*$f \in$ **FP** and for all $x \in \Sigma^*$, $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

Easy Fact: If $L_1 \leq_p L_2$, then $L_1 \xrightarrow{p} L_2$, and therefore if $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.

We showed earlier how GKS can be regarded as a "special case" of $SDPDS$. We can formalize this assertion by stating:

Claim: GKS $\xrightarrow{p}$ SDPDS, and
GKD $\leq_p$ SDPDD.

To show GKD $\leq_p$ SDPDD, consider an input x for GKD. Assume x is an instance of GKD, $x = \langle (w_1, g_1), \cdots, (w_m, g_m), W, B \rangle$.

(It is easy to compute if x is an instance of GKD, and if it isn't, we just let $f(x)$ be some trivial string not in SDPDD.)
Then we let $f(x) = \langle (W, g_1, w_1), \cdots, (W, g_m, w_m), B \rangle$.
It is easy to see that f is computable in polynomial time,
and that for all $x \in \Sigma^*$, $x \in GKD \Leftrightarrow f(x)$ SDPDD.

A corollary of all this is that if Scheduling With Deadlines, Profits and Durations is polynomial-time computable, then the General Knapsack problem is polynomial-time computable. We will see later that the theory of **NP**-completeness implies that the converse is true as well.

It is important to realize that both $\xrightarrow{p}$ and $\leq_p$ are *transitive*. We will prove this for $\leq_p$.

Theorem 2 *Let $L_1, L_2, L_3 \subseteq \Sigma^*$. If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.*

Proof:
Say that $L_1 \leq_p L_2$ via function $f_1 : \Sigma^* \to \Sigma^*$, where Turing machine M_1 computes f_1 in time $O(n^c)$ on inputs of length n; say that $L_2 \leq_p L_3$ via function $f_2 : \Sigma^* \to \Sigma^*$, where Turing machine M_2 computes f_2 in time $O(n^d)$ on inputs of length n.

Define $h : \Sigma^* \to \Sigma^*$ by $h(x) = f_2(f_1(x))$. Clearly for all $x \in \Sigma^*$,
$x \in L_1 \Leftrightarrow f_1(x) \in L_2 \Leftrightarrow f_2(f_1(x)) \in L_3 \leftrightarrow h(x) \in L_3$.

We can compute h by a machine M in polynomial time as follows: Let x be the input, $|x| = n$. M begins by computing $y = f_1(x)$ by running M_1 on x. This will take time $O(n^c)$, and y will have length $O(n^c)$. M then runs M_2 on y to compute $f_2(y) = h(x)$, in time $O(|y|^d) = O((n^c)^d) = O(n^{cd})$. The total time for M is $O(n^c) + O(n^{cd}) = O(n^{cd})$, a polynomial in n. $\square$