Design and Analysis of Algorithms

Lecture Notes

Learning objectives and prerequisites

On completing this course, students should be able to:

- Identify problems solvable with greedy algorithms, design and prove the correctness of such algorithms, and provide asymptotic running time analyses for a variety of given algorithms.
- Recognize problems to which divide and conquer or dynamic programming approaches may apply, design algorithms using these approaches, and analyze their computational efficiency.
- Apply randomization to produce tractable algorithms for several specific computationally challenging problems.
- Determine whether certain problems are computationally intractable, and use reductions to known problems to prove intractability.
- Use approximation algorithms to efficiently produce near-optimal solutions for intractable problems, and bound how close these algorithms are to being optimal.

We assume that everyone is familiar with elementary data structures, probability, sorting, and basic graph terminology, including the concepts of depth-first search and breadth-first search, Prim's and Kruskal's algorithms for finding minimum spanning trees, and Dijkstra's and Bellman-Ford's algorithms for finding shortest paths. The lectures involve the analysis of algorithms at a fairly mathematical level. We expect everyone to be comfortable reading and writing proofs.

Stable matching

The Stable Matching Problem originated, in part, in 1962, when David Gale and Lloyd Shapley, two mathematical economists, asked whether one could design a college admissions process, or a job recruiting process that is self-enforcing. More precisely: Given a set of preferences among employers and applicants, can we assign applicants to employers so that, for every employer E, and every applicant A who is not assigned to E, at least one of the following two things holds:

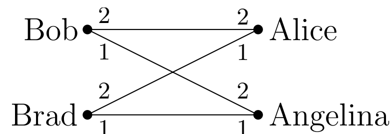
- E prefers all of its accepted applicants over A, or
- A prefers her current assignment over working for employer E.

If this condition holds, the outcome is stable: individual self-interest will prevent any applicant-employer deal from being made behind the scenes.

Gale and Shapley proceeded to develop a striking algorithmic solution to this problem, which we will discuss here.

Stable marriage

We are going to investigate a simplified variant of the assignment problem described above. Here, we want to match boys and girls, each of whom has a preference order over possible mates of the opposite sex. The preferences don't have to be symmetric. For example, Alice might really like Brad, but Brad has the hots for Angelina. Suppose Angelina also likes Brad more than Bob, but Bob really likes Angelina.



Now, suppose we were to pair Brad with Alice and Bob with Angelina. Well, that would lead to a very dicey situation! Suffice it to say that, pretty

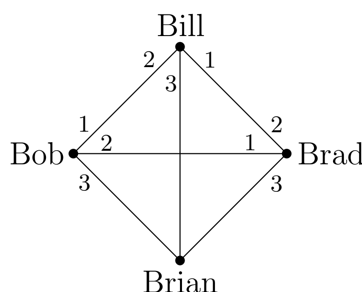
soon, Brad and Angelina are likely to start spending late nights doing algorithms homework together. The main problem is that Brad and Angelina each prefer each other to their assigned partners in the matching. In such a circumstance, we say that Brad and Angelina form a rogue couple. More precisely, we'll say that given a matching M, a boy b and a girl g are a rogue couple for M if b and g prefer each other to their partners in M.

Obviously, the existence of rogue couples is not a good thing if you are making matchings, since they lead to instability. Thus, we say that a matching is stable if there are no rogue couples.

We are going to assume that preferences do not change over time. That is, we are not modeling situations in which preferences evolve. Preferences are known at the start and remain unchanged.

Our main goal is to find a perfect matching that is stable. In the above example, a possible stable perfect matching is to pair Brad with Angelina, and Bob with Alice. Bob and Alice may not be very happy, but no rogue couple is possible, and so the matching is stable. That is because neither Brad nor Angelina prefers anyone to each other, so even though Alice and Bob are not happy with each other, no one else will form a rogue couple with either of them.

In general, it is not clear that there is always a stable matching for any number of people and set of preference orders. In fact, if you allow boys to prefer boys and girls to prefer girls, then there are examples in which no stable matching exists. Here is a simple unisex example in which a stable matching is not possible. The idea is to create a "love triangle" with a fourth person who is everyone's last choice:



It turns out that the fourth person's preferences do not even matter. Let's see why no stable matching exists. We'll prove this by contradiction. Assume, for contradiction, that there is a stable matching. Then two members of the love triangle must be matched. Without loss of generality (by symmetry), assume that Bill is matched with Bob. Then the other pair must be Brad matched with Brian. But now there is a rogue couple since Bill likes Brad

best and Brad prefers Bill to Brian. Thus, Bill and Brad form a rogue couple. Consequently, no stable matching is possible.

This proposition is not very surprising — obtaining a stable matching can be tricky. What is surprising, however, is that a stable matching can always be found in the two-sided case, that is, when boys are only allowed to pair with girls, and vice versa.

Let us now formalize the statement of the problem we are discussing.

The setting:

- There are n boys and n girls; we assume the number of boys and girls is the same.
- Each boy has a ranked preference list of the girls.
- Each girl has a ranked preference list of the boys.
- The lists are complete and have no ties; each boy ranks every girl and each girl ranks every boy.

The goal:

Pair each boy with a unique girl so that no rogue couples exist. In other words, find a perfect matching in which every boy and girl is paired one-to-one, with no potential instability.

Gale-Shapley algorithm

It turns out that finding a good way to pair up the boys and girls is a tricky problem. The best approach is to use the Gale-Shapley algorithm, named after the mathematicians David Gale and Lloyd Shapley, who devised it in 1962. Note that, in 2012, the Nobel Prize in Economics was awarded to Lloyd Shapley and Alvin Roth "for the theory of stable allocations and the practice of market design."

Here is the method for pairing everyone up. The "mating ritual" takes place over several days. The idea is that each boy proposes to girls one by one, in order of preference, crossing them off his list whenever he is rejected. A more detailed specification follows:

Initial condition:

Each of the n boys has an ordered list of the n girls according to his preferences. Similarly, each girl has an ordered list of the n boys according to her preferences.

Each day

- Morning:
  - Each girl stands on her balcony.
  - Each boy stands under the balcony of his favorite girl whom he has not yet crossed off his list and serenades her. If there are no girls left on his list, he stays home and does algorithms homework.
- Afternoon:
  - Girls who have at least one suitor say to their favorite among the suitors that day: "Maybe I will marry you — come back tomorrow!"
  - To all others, they say "No, I will never marry you!"
- Evening:
  - Any boy who hears "No" crosses that girl off his list.

Termination condition:

If there is a day when every girl has at most one suitor, the ritual stops and each girl marries her current suitor (if any).

Now let's show that the algorithm works. We need to demonstrate that:

- The Gale-Shapley algorithm terminates.
- The Gale-Shapley algorithm terminates quickly.
- At termination, there are no rogue couples.
- Everyone is married.

We will also analyze the fairness of the protocol: Is it better for boys or for girls?

Let's start by showing that this algorithm terminates:

Theorem 1. The Gale-Shapley algorithm terminates within $n^2 + 1$ days.

Proof. We prove this theorem by contradiction. Suppose, for contradiction, that the Gale-Shapley algorithm does not terminate within $n^2 + 1$ days. Consider any day on which the algorithm does not terminate. On such a day, some boy must cross a girl off his list. Why? If the algorithm does not terminate, at least one girl must have two or more suitors. In this case, she rejects at least one boy, and that boy crosses her name off his list. Therefore, if the Gale-Shapley algorithm doesn't terminate in n^2+1 days, at least n^2+1 names must have been crossed off in total. However, at the start, each boy's list contains n names, so the total number of names across all the lists is n^2; thus it is impossible to cross off more names than this total. This is a contradiction, consequently, the algorithm must terminate within $n^2 + 1$ days.

This is a typical proof technique in Computer Science used to bound the running time of an algorithm. We show that the algorithm always makes progress by some measure. Since there is only a finite amount of progress to make, it must eventually terminate. Here the measure is the total number of names on the union of all the lists of the boys.

Next, we'll prove that everyone gets married by the Gale-Shapley algorithm, but first we need a couple of lemmas.

Lemma 1. If a boy marries, then he courted every girl he liked better.

Proof. In the Gale-Shapley algorithm, boys cross girls off their lists one at a time in order of preference, starting with the girl they like most. A boy marries the girl (if any) whom he is courting at termination. Thus, if a boy marries, he marries the least-preferred girl among those he courted. Tough luck for the boy, but at least he prefers her to all the girls he never courted.

Lemma 2. If a boy never marries, then he courted every girl.

Proof. By Theorem 1, the Gale-Shapley algorithm terminates. At the time of termination, this boy is not courting anyone. How can that be? If he is at home doing his algorithms homework, then he must have crossed off every girl on his list. Therefore, he has courted every girl.

Lemma 3. A girl marries her favorite among her suitors.

Proof. A girl only rejects a boy when a more preferred one comes along and always keeps her most-preferred suitor among those she has seen so far.

This also implies:

Lemma 4. If a girl is ever courted, she gets married.

Proof. Once a girl has a suitor, she keeps him until she trades up.

Now we can prove that everyone gets married:

Theorem 2. Everyone is married in the Gale-Shapley algorithm.

Proof. We prove this by contradiction. Suppose, for contradiction, that some boy b is not married. By Lemma 2, boy b has courted every girl. Therefore, every girl has been courted. By Lemma 4, every girl who is ever courted gets married. Hence, every girl is married. Since there are equal numbers of boys and girls, it follows that every boy — including b — must be married. This contradicts our assumption. Therefore, everyone is married, as claimed.

Next we prove the main result, namely that the Gale-Shapley algorithm always produces stable marriages.

Theorem 3. The Gale-Shapley algorithm produces stable marriages.

Proof. Suppose, for contradiction, that there exists a rogue couple formed by b and g. Let b be married to g' and let g be married to b' in the Gale-Shapley matching. If b is married to g', but prefers g to g', then by Lemma 1, boy b must have courted g before g'. At that time, g rejected b. By Lemma 3, girl g only rejects a boy in favor of someone she prefers more. Therefore, g prefers b' to b. This means that g does not prefer b to her assigned partner, so b and g does not form a rogue couple, a contradiction. Hence, no rogue couples exist, and the matching is stable.

Well, who do you think is better off in the Gale-Shapley algorithm? In other words, who has the power, the proposers or the acceptors? Since the girls marry their favorite from among their suitors, and the boys end up with the least-preferred girls they ever court, it might seem reasonable to assume that the girls do best. It seems hard to answer this question formally, especially since it isn't even clear what we mean by "doing better". But, in fact, we can show in a very precise and formal way that the algorithm is heavily biased toward the boys. To formalize this, we need to define the set of realistic potential mates.

Let S be the set of all stable matchings for a given instance of the stable marriage problem. Since the Gale-Shapley algorithm produces a stable

matching, we know that $S \neq \emptyset$. For each person p, we define the realm of possibility for p as follows: a person q of the opposite sex is within the realm of possibility for p if and only if there exists a stable matching in which p is married to q.

Some mates may simply be out of the question, since no stable pairing is possible if you marry them. In our initial example, Brad is not a realistic match for Alice, because if they are paired, Brad and Angelina will form a rogue couple – so there is no stable matching in which Brad is paired with Alice.

Definition. A person's optimal mate is their most-preferred partner from the realm of possibility. Analogously, a person's pessimal mate is their least-preferred partner from the realm of possibility.

Both an optimal and a pessimal mate must exist, since we know there is at least one stable matching — namely the one produced by the Gale-Shapley algorithm.

Ok, now here is a pair of shocking results:

Theorem 4. The Gale-Shapley algorithm pairs every boy with his optimal mate!

Theorem 5. The Gale-Shapley algorithm pairs every girl with her pessimal mate!

This is too hard to believe, so let's prove it.

Proof of Theorem 4. Assume, for contradiction, that some boy does not get his optimal girl (that is, his favorite girl within the realm of possibility). Let b be the first (in time) boy who gets rejected by his optimal girl g (resolving ties arbitrarily). Define b' to be the boy who causes g to reject b in the Gale-Shapley algorithm. Then g prefers b' to b.

Since b is the first boy to be rejected by his optimal mate in the Gale-Shapley algorithm, b' has not yet been rejected by his optimal mate when he is courting g. Thus, b' likes g at least as much as he likes his optimal mate g^* (now g and g^* might be the same person).

Let M be a stable matching in which b marries g. Such a matching exists since g is in the realm of possibility for b. By assumption, M is not produced by the Gale-Shapley algorithm. Let g' be the mate of b' in M. By definition, b' likes g^* at least as much as g' (again, they might be the same person), so b' prefers g to g', since he likes g at least as much as g^*, whom he likes at

least as much as g'. (Note that g can't be the same person as g'.) Therefore, b' and g form a rogue couple, which contradicts the fact that M is a stable matching.

Now let's show that the girls get their pessimal mate.

Proof of Theorem 5. Suppose, for contradiction, that there exists a stable matching M in which there is a girl g who fares worse than in the Gale-Shapley algorithm. Let b be the mate of g in the Gale-Shapley algorithm, and let b' be the mate of g in M. Then g prefers b to b', since she fares worse in M than in the Gale-Shapley algorithm. Let g' be the mate of b in M.

By Theorem 4, the Gale-Shapley algorithm gives b an optimal mate, so b prefers g to g'. Therefore, b and g form a rogue couple in M, which is a contradiction.

Uniqueness

For obvious reasons, the stable matching generated by the Gale-Shapley algorithm is called boy-optimal and girl-pessimal. If the roles of the sexes in the algorithm are interchanged, i.e., if girls court boys, then the resulting stable matching is analogously girl-optimal and boy-pessimal. It may happen that the boy-oriented and the girl-oriented versions of the algorithm yield the same stable matching, in which case it is immediate, by combining the optimality and pessimality properties, that this matching is unique.

Number of stable matchings

The following example shows that a systematic search for all stable matchings can lead us to consider an exponential number of possible cases. Let n be an even number and assume we have n boys $b_1, b_2, \ldots, b_n$ and n girls $g_1, g_2, \ldots, g_n$ with preferences as given below.

$$\begin{aligned}
b_1 &\rightarrow (g_1, g_2, g_3, g_4, \ldots, g_{n-1}, g_n) \\
b_2 &\rightarrow (g_2, g_1, g_3, g_4, \ldots, g_{n-1}, g_n) \\
b_3 &\rightarrow (g_3, g_4, g_5, g_6, \ldots, g_1, g_2) \\
b_4 &\rightarrow (g_4, g_3, g_5, g_6, \ldots, g_1, g_2) \\
&\vdots \\
b_{n-1} &\rightarrow (g_{n-1}, g_n, g_1, g_2, \ldots, g_{n-3}, g_{n-2}) \\
b_n &\rightarrow (g_n, g_{n-1}, g_1, g_2, \ldots, g_{n-3}, g_{n-2})
\end{aligned}$$

$$
\begin{aligned}
g_1 &\to (b_2, b_3, b_4, \ldots, b_n, b_1) \\
g_2 &\to (b_3, b_4, b_5, \ldots, b_1, b_2) \\
g_3 &\to (b_4, b_5, b_6, \ldots, b_2, b_3) \\
g_4 &\to (b_5, b_6, b_7, \ldots, b_3, b_4) \\
&\vdots \\
g_{n-1} &\to (b_n, b_1, b_2, \ldots, b_{n-2}, b_{n-1}) \\
g_n &\to (b_1, b_2, b_3, \ldots, b_{n-1}, b_n)
\end{aligned}
$$

Suppose that each pair of boys, b_1 and b_2, b_3 and b_4, $\ldots$ marries either their first or second choice. One can construct $2^{n/2}$ different matchings in this way. We claim that each matching so obtained is stable. Indeed, when two boys marry their second choice, they cannot obtain their first choice because they are the least preferred by the girls in question.

Optimality

Recall that the Gale-Shapley algorithm finds a stable matching with an extreme property that every boy gets his best possible partner among all stable matchings. In this sense, the matching is boy-optimal. Of course, if we exchange the roles of boys and girls, the resulting stable matching is girl-optimal. Unfortunately, by the nature of stable matchings, the boy-optimal stable matching is simultaneously the girl-pessimal stable matching, that is, every girl gets her worst possible partner. Conversely, the girl-optimal stable matching is simultaneously the boy-pessimal stable matching. Hence, it is natural to seek a matching that is not only stable but also "good" according to some criterion. There are many optimization criteria for the quality of stable matchings, here we introduce three of them.

Let $p_b(g)$ denote the position of girl g in boy b's preference list, and, similarly, let $p_g(b)$ denote the position of boy b in girl g's preference list. For a stable matching M, define the regret cost $r(M)$ to be

$$r(M) = \max_{(b,g)\in M} \max\{p_b(g), p_g(b)\},$$

the egalitarian cost $c(M)$ to be

$$c(M) = \sum_{(b,g)\in M} p_b(g) + \sum_{(b,g)\in M} p_g(b),$$

and the sex-equalness cost $d(M)$ to be

$$d(M) = \left| \sum_{(b,g)\in M} p_b(g) - \sum_{(b,g)\in M} p_g(b) \right|.$$

The minimum regret stable marriage problem, the minimum egalitarian stable marriage problem, and the sex-equal stable marriage problem, respectively, ask for a stable matching M that minimizes $r(M)$, $c(M)$, and $d(M)$. Note that the number of stable matchings for a given instance grows exponentially in general. Nevertheless, Gusfield proposed a polynomial-time algorithm for the first problem, and Irving, Leather, and Gusfield did so for the second. In contrast, the sex-equal stable matching problem is NP-hard.

Greedy algorithms

It is hard, if not impossible, to define precisely what is meant by a greedy algorithm. An algorithm is greedy if it builds a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. When a greedy algorithm succeeds in solving a nontrivial problem optimally, it typically implies something interesting and useful about the structure of the problem itself; there is a local decision rule that one can use to construct optimal solutions. It's easy to invent greedy algorithms for almost any problem; finding cases in which they work well, and proving that they do so, is the interesting challenge.

Scheduling with minimal lateness

Consider a situation in which we have a single resource and a set of n tasks $a_1, a_2, \ldots, a_n$, each requesting the resource for some period of time. Assume that the resource is available starting at time 0. The resource can process at most one task at a time, and once a task is started, it occupies the resource until it is completed. Each request a_i has a deadline d_i and requires a contiguous time interval of length t_i to process, but it can be scheduled at any time before the deadline. Each accepted request must be assigned a time interval of length t_i, and different requests must be assigned nonoverlapping intervals.

There are many objective functions we might seek to optimize in this situation, and some are computationally much more difficult than others. Here we consider a very natural goal that can be optimized by a greedy algorithm. Suppose that we plan to schedule all requests, but we are allowed to let certain requests run late. Beginning at the overall start time 0, we assign each request a_i a time interval of length t_i; denote this interval by $[s_i, f_i]$, where $f_i = s_i + t_i$.

We say that request a_i is late if it misses its deadline, that is, if $f_i > d_i$. The lateness of such a request a_i is defined to be $l_i = f_i - d_i$. If the request is not late, we define $l_i = 0$. The goal in our optimization problem is to

schedule all requests using nonoverlapping intervals so as to minimize the maximum lateness $L = \max_i l_i$.

Assume that the tasks are sorted in increasing order of their deadlines (if they are not, first sort them):

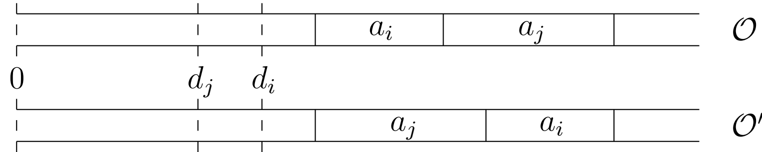
$$d_1 \leqslant d_2 \leqslant \cdots \leqslant d_n.$$

Schedule the tasks in this order as follows: let $s_1 = 0$ be the starting time and $f_1 = s_1 + t_1$ be the finishing time of task a_1, let $s_2 = f_1$ be the starting time and $f_2 = s_2 + t_2$ be the finishing time of a_2, and so on.

We now prove that this schedule is optimal. Let $\mathcal{P}$ denote the schedule produced by this algorithm. Note that this schedule has no idle time, i.e., the resource is never idle while tasks remain to be scheduled.

Evidently, there is an optimal schedule with no idle time as well. Let $\mathcal{O}$ denote such an optimal schedule. If $\mathcal{O} = \mathcal{P}$, then we are done. Now assume that $\mathcal{O} \neq \mathcal{P}$. Then $\mathcal{O}$ must contain two tasks a_i and a_j scheduled consecutively with $i > j$.

Consider the schedule $\mathcal{O}'$ obtained by swapping tasks a_i and a_j in $\mathcal{O}$.



What can we say about the maximal lateness in $\mathcal{O}'$? Only the schedules of tasks a_i and a_j have changed, therefore the lateness of every other task remains the same after the swap. Task a_j is completed earlier after the swap, so its lateness evidently doesn't increase.

Now consider task a_i. Let f_j denote the finishing time and l_j denote the lateness of a_j in schedule $\mathcal{O}$. Then the finishing time of a_i in schedule $\mathcal{O}'$ is $f_i' = f_j$. Let l_i' denote the lateness of a_i in $\mathcal{O}'$.

If $l_i' = 0$, that is, a_i is not late in schedule $\mathcal{O}'$, then it is also not late in schedule $\mathcal{O}$, since it finishes earlier there. On the other hand, if $l_i' > 0$, then, taking into account that $d_i \geqslant d_j$, we have

$$l_i' = f_i' - d_i = f_j - d_i \leqslant f_j - d_j = l_j.$$

This means that the lateness of a_i in schedule $\mathcal{O}'$ cannot exceed the lateness of a_j in schedule $\mathcal{O}$. Consequently, the maximal lateness in $\mathcal{O}'$ cannot exceed the maximal lateness in $\mathcal{O}$.

The correctness of the algorithm now follows easily. Starting from an optimal schedule $\mathcal{O}$ we can obtain the schedule $\mathcal{P}$ by repeatedly swapping consecutive tasks at most $\binom{n}{2}$ times (cf. bubble sort). Since the maximal lateness does not increase in any step, $\mathcal{P}$ must also be an optimal schedule.

The algorithm runs in $O(n \log n)$ time.

Scheduling without lateness

Again, we are given a set of n tasks $\mathcal{A} = \{a_1, a_2, \dots, a_n\}$ that can be completed only using a shared resource. The resource is available starting at time 0. The resource can process at most one task at a time, and once a task is started, it occupies the resource until it is completed. For each task a_i, we are given its processing time t_i and its deadline d_i, by which the task must be completed.

We want to devise an efficient algorithm that finds a subset of maximum cardinality $\{a_{i_1}, a_{i_2}, \dots, a_{i_k}\} \subseteq \mathcal{A}$, along with starting times $s_{i_j} \geqslant 0$ and finishing times f_{i_j} for each task a_{i_j}, such that

- $f_{i_j} - s_{i_j} = t_{i_j}$ and $f_{i_j} \leqslant d_{i_j}$, for all $1 \leqslant j \leqslant k$.
- The intervals $[s_{i_j}, f_{i_j}]$ and $[s_{i_h}, f_{i_h}]$ are pairwise nonoverlapping, for all $1 \leqslant j < h \leqslant k$.

The first criterion ensures that no task a_{i_j} is late, while the second guarantees that the resource is never used by more than one task at a time and that tasks are processed without interruptions.

Assume that the tasks are sorted in increasing order of their deadlines (if they are not, first sort them):

$$d_1 \leqslant d_2 \leqslant \cdots \leqslant d_n.$$

We maintain a set H containing an optimal subset of the tasks. Initially $H = \emptyset$. For each $1 \leqslant i \leqslant n$, in increasing order, add a_i to H. If the total processing time of tasks in H exceeds d_i, i.e.,

$$\sum_{a_j \in H} t_j > d_i,$$

remove from H the task with the longest processing time. Finally, schedule the tasks in H in increasing order of their deadlines, without idle time, starting at time 0. Let $\mathcal{H}$ denote this schedule. Note that, in schedule $\mathcal{H}$, no task is late, since the algorithm ensures that the cumulative processing time never exceeds the current task's deadline.

We prove by mathematical induction on the number of tasks that $\mathcal{H}$ is an optimal solution to the problem. The base case $n = 1$ is trivial. Next, assume that $n > 1$ and that the statement holds for $n - 1$ tasks.

If $\mathcal{H}$ contains all tasks, then it is evidently an optimal solution to the problem. Now, assume that there exists a task that is not included in $\mathcal{H}$. Let a_k be the first task removed from H when adding some task a_i to H.

We show that the problem has an optimal solution $\mathcal{O}$ where tasks are scheduled in increasing order of their deadlines, without idle time, starting at time 0, and which does not include task a_k. Clearly, there exists an optimal solution in which tasks are scheduled in increasing order of their deadlines, without idle time, starting at time 0. Consider such an optimal solution $\mathcal{O}'$, and suppose it includes task a_k. Since

$$\sum_{j=1}^{i} t_j > d_i,$$

there must exist a task a_l among tasks $a_1, a_2, \ldots, a_i$ that is not in $\mathcal{O}'$. Let $\mathcal{O}$ be the schedule obtained from $\mathcal{O}'$ by replacing task a_k with task a_l, and then scheduling all tasks in increasing order of their deadlines, without idle time, starting at time 0. Observe that no task among $a_1, a_2, \ldots, a_i$ in $\mathcal{O}$ is late, since the choice of i guarantees that this property holds when all tasks $a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_i$ are scheduled in increasing order of their deadlines, without idle time, starting at time 0. On the other hand, no task among $a_{i+1}, a_{i+2}, \ldots, a_n$ in $\mathcal{O}$ is late either, since none of them is scheduled later than in $\mathcal{O}'$, because $t_k \geqslant t_l$. Thus, $\mathcal{O}$ is also an optimal solution to the problem.

Obviously, the number of tasks in $\mathcal{H}$ cannot exceed the number of tasks in $\mathcal{O}$, since $\mathcal{O}$ is an optimal solution to the problem. On the other hand, by executing the greedy algorithm on the set of tasks $\mathcal{A} \setminus \{a_k\}$, we also obtain the schedule $\mathcal{H}$, which, by the induction hypothesis, is an optimal solution for the set $\mathcal{A} \setminus \{a_k\}$. Since schedule $\mathcal{O}$ contains tasks only from the set $\mathcal{A} \setminus \{a_k\}$, the number of tasks in $\mathcal{O}$ cannot exceed the number of tasks in $\mathcal{H}$. It follows that the number of tasks in $\mathcal{H}$ is equal to the number of tasks in $\mathcal{O}$, and thus $\mathcal{H}$ is an optimal solution to the original problem.

Storing the tasks in H in a max-heap sorted by their processing time, inserting the next task and potentially removing the task with the maximal processing time can be done in $O(\log n)$ time. Checking whether the deletion is necessary takes $O(1)$ time if an auxiliary variable is used to store the total processing time for the tasks stored in the heap. Performing this operation for every task one after another takes $O(n \log n)$ time. The initial sorting can also be done in $O(n \log n)$ time, thus, the total running time of the algorithm

is $O(n \log n)$.

Clustering

Clustering arises whenever one has a collection of objects — for example, a set of photographs, documents, or microorganisms — that one is trying to classify or organize into coherent groups. In such situations, it is natural to first consider measures of similarity or dissimilarity between pairs of objects. One common approach is to define a distance function on the objects, interpreting larger distances as indicating less similarity. For instance, the distance between two images in a video stream could be defined as the number of corresponding pixels where their intensity values differ by at least some threshold.

Now, given a distance function on the objects, the clustering problem seeks to divide them into groups so that, intuitively, objects within the same group are "close together", and objects in different groups are "far apart". From this general notion, the field of clustering branches into many technically distinct approaches, each formalizing what constitutes a "good" clustering in a particular way.

More concretely, we are given a positive integer k, a set of $n \geqslant k$ objects $\mathcal{U} = \{p_1, p_2, \ldots, p_n\}$, and a distance function d on $\mathcal{U}$. We allow any distance function d that satisfies the following properties:

- $d(p_i, p_j) \geqslant 0$,
- $d(p_i, p_j) = 0$ if and only if $i = j$,
- $d(p_i, p_j) = d(p_j, p_i)$.

A family $\{C_1, C_2, \ldots, C_k\}$ of k non-empty subsets of $\mathcal{U}$ is called a k-clustering of $\mathcal{U}$ if

- $C_1 \cup C_2 \cup \cdots \cup C_k = \mathcal{U}$,
- $C_i \cap C_j = \emptyset$, if $i \neq j$.

Define the spacing of a k-clustering $\{C_1, C_2, \ldots, C_k\}$ as

$$\min\{d(p_i, p_j) \mid p_i \in C_s,\ p_j \in C_t,\ s \neq t\},$$

i.e., the minimum distance between any pair of objects lying in different clusters.

Since we want objects in different clusters to be far apart, a natural goal is to find the k-clustering with maximum possible spacing. The question now becomes the following: There are exponentially many different k-clusterings of a set $\mathcal{U}$, how can we efficiently find the one with maximum spacing?

Introduce a weighted complete graph whose vertices represent the objects, and where the weight of an edge between two vertices is the distance between the corresponding objects. Proceed exactly as in Kruskal's minimum spanning tree algorithm on this graph, but stop once k connected components are obtained. This is equivalent to constructing the full minimum spanning tree as Kruskal's algorithm would normally produce, then removing the $k-1$ most expensive edges (the ones we would not actually add to maintain k components). The resulting vertex sets of the connected components define the k-clustering.

We prove that the spacing of the k-clustering $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ produced by the above algorithm is maximal. The spacing of $\mathcal{C}$ is obviously the weight of the $(k-1)^{\text{th}}$ most expensive edge of the minimum spanning tree produced by Kruskal's algorithm. Let d^* denote this weight.

Now consider an arbitrary k-clustering $\mathcal{C}' = \{C'_1, C'_2, \ldots, C'_k\}$ of $\mathcal{U}$, different from $\mathcal{C}$. Since $\mathcal{C}$ and $\mathcal{C}'$ are different k-clusterings of the same set $\mathcal{U}$, there must exist a set $C_r \in \mathcal{C}$ that is not a subset of any of the k sets in $\mathcal{C}'$. This implies that C_r contains two objects p_i and p_j that belong to different clusters in $\mathcal{C}'$, say, $p_i \in C'_s$ and $p_j \in C'_t$, where $s \neq t$. Since p_i and p_j belong to the same set C_r in $\mathcal{C}$, Kruskal's algorithm must have added all edges of some path P connecting p_i and p_j to their connected component before termination. Each edge on P therefore has weight at most d^*.

Recall that $p_i \in C'_s$ but $p_j \notin C'_s$. Let p' denote the first vertex on P that is not in C'_s, and let p be the vertex immediately preceding p' on P. We have just argued that $d(p, p') \leqslant d^*$. But p and p' belong to different clusters of $\mathcal{C}'$, so the spacing of $\mathcal{C}'$ is at most $d(p, p') \leqslant d^*$.

This completes the proof.

Divide and conquer algorithms

Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem on each part recursively, and then combines the solutions to these subproblems into an overall solution.

Analyzing the running time of a divide and conquer algorithm typically involves solving a recurrence relation that expresses the running time recursively in terms of the running time on smaller instances.

Collaborative filtering

We will consider a problem that arises in the analysis of rankings, which is becoming increasingly important in a number of modern applications. For example, many websites use a technique known as collaborative filtering, in which they try to match your preferences (for books, movies, restaurants, etc.) with those of other people on the Internet. Once the website identifies people with "similar" tastes to yours — based on a comparison of how you and they rate various items — it can recommend new items that these other people have liked.

A core issue in applications like this is the problem of comparing two rankings. Suppose you rank a set of n movies, and the collaborative filtering system consults its database to find other people with "similar" rankings. But what is a good numerical measure of similarity between two rankings? Clearly, an identical ranking is very similar, while a completely reversed ranking is very different. We want a measure that interpolates through the middle, reflecting partial agreement.

Consider comparing your ranking with a stranger's ranking of the same set of n movies. A natural method is to label the movies from 1 to n according to your ranking, then order these labels according to the stranger's ranking, and see how many pairs are "out of order".

More concretely, we formulate the problem as follows: We are given an array $A[1:n]$ of n pairwise distinct numbers. We want a measure of how far

the array is from being sorted in ascending order. The measure should be 0 if $A[1] < A[2] < \ldots < A[n]$, and should increase as the array becomes more scrambled. A natural way to quantify this notion is by counting the number of inversions. We say that two indices $i < j$ form an inversion if $A[i] > A[j]$. Our goal is to determine the number of inversions in the array $A[1 : n]$.

The simplest algorithm to count inversions is straightforward: Look at every pair of indices $1 \leqslant i < j \leqslant n$ and check whether they form an inversion. This takes $O(n^2)$ time. We will show how to count the number of inversions much more efficiently.

First consider the insertion sort.

```
InsertionSort(A[1:n])
for j=2 to n do
  key=A[j]
  i=j-1
  while i>0 and A[i]>key do
    A[i+1]=A[i]
    i=i-1
  A[i+1]=key
```

Not counting the number of iterations of the while loops, the algorithm runs in $O(n)$ time. Let I be the number of inversions in the array. In each iteration of the while loops, exactly one inversion disappears. When the algorithm terminates, no inversions remain. This implies that the number of iterations of the while loops is exactly I. Therefore, the total running time of the algorithm is $O(n + I)$. Since the largest possible value of I is $\binom{n}{2}$, the algorithm is still quadratic in the worst case. However, if I is small, the algorithm is much faster.

To obtain a highly efficient general algorithm, we slightly modify merge sort in the following way. The procedure must initially be invoked with the parameter $A[1 : n]$.

```
NumberOfInversions(A[p:r])
inv=0
if p<r then
  q=[(p+r)/2]
  inv=inv+NumberOfInversions(A[p:q])
  inv=inv+NumberOfInversions(A[q+1:r])
  inv=inv+NumberOfInversionsWithMerge(A[p:q],A[q+1:r])
return inv
```

```
NumberOfInversionsWithMerge(A[p:q],A[q+1:r])
n_1=q-p+1
n_2=r-q
for i=1 to n_1 do
  L[i]=A[p+i-1]
for j=1 to n_2 do
  R[j]=A[q+j]
L[n_1+1]=INFINITY
R[n_2+1]=INFINITY
i=1
j=1
inv=0
for k=p to r do
  if L[i]<R[j]
    then
      A[k]=L[i]
      i=i+1
    else
      A[k]=R[j]
      inv=inv+n_1-i+1
      j=j+1
return inv
```

First, we divide the array $A[1:n]$ into two subarrays of roughly equal size, $A[1:q]$ and $A[q+1:n]$, where $q=[(n+1)/2]$, and we recursively determine the number of inversions in each subarray.

Then, we count the inversions (i',j'), where $1 \leqslant i' \leqslant q$ and $q+1 \leqslant j' \leqslant n$ using the procedure `NumberOfInversionsWithMerge`. Note that, when this procedure is called, the subarrays $A[1:q]$ and $A[q+1:n]$ are already sorted.

To prove the correctness of the algorithm, it is enough to prove the correctness of the procedure `NumberOfInversionsWithMerge`.

Suppose that, during the execution of the algorithm, we compare the value $A[i']$ in $A[p:q]$ with the value $A[j']$ in $A[q+1:r]$ where $p \leqslant i' \leqslant q$ and $q+1 \leqslant j' \leqslant r$. Also, suppose that we have already discovered all inversions whose first component lies between p and $i'-1$ and whose second component lies between $q+1$ and $j'-1$.

- If $A[i'] < A[j']$, then, taking into account that $A[j'] < A[j'+1] < \cdots < A[n]$, no new inversion is discovered, whose first component is i', so the number of inversions does not increase in this case.

- On the other hand, if $A[i'] > A[j']$, then, taking into account that

$A[q] > \cdots > A[i'+1] > A[i']$, we discover the new inversions (i',j'), $(i'+1,j'),\ldots,(q,j')$, so the number of inversions increases by $q-i'+1$ in this case.

This completes the proof.

The overall running time of the algorithm is $O(n \log n)$, just as in standard merge sort.

Investment

We are looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, the stock has a price $A[i]$ per share (we assume, for simplicity, that the price is fixed during each day). We would like to know: How should we choose a day i on which to buy the stock and a later day $j \geqslant i$ on which to sell it, if we want to maximize the profit $A[j] - A[i]$ per share?

Mathematically, we are given an array $A[1:n]$ of n positive numbers. We want to devise an efficient algorithm that determines two indices $1 \leqslant i \leqslant j \leqslant n$ such that $A[j] - A[i]$ is as large as possible.

We apply a divide and conquer strategy. Divide the array $A[1:n]$ into the subarrays $A[1:m]$ and $A[m+1:n]$ where $m = [(n+1)/2]$.

(1) Find recursively indices $1 \leqslant i \leqslant j \leqslant m$ for which $A[j]-A[i]$ is maximal in $A[1:m]$.

(2) Find recursively indices $m+1 \leqslant i \leqslant j \leqslant n$ for which $A[j] - A[i]$ is maximal in $A[m+1:n]$.

(3) Find indices $1 \leqslant i \leqslant m < j \leqslant n$ for which $A[j] - A[i]$ is maximal. In this case $A[i]$ must be a minimum element in the subarray $A[1:m]$, and $A[j]$ must be a maximum element in the subarray $A[m+1:n]$.

The solution to the original problem is given by the case among these three that yields the largest profit.

Procedure `MaximalIncrease(A[l:r]` returns indices $l \leqslant i \leqslant j \leqslant r$ for which $A[j] - A[i]$ is maximal in the subarray $A[l:r]$. The procedure must initially be invoked with the parameter $A[1:n]$.

```
MaximalIncrease(A[l:r])
if l=r then return (l,l)
m=[(l+r)/2]
(x,x')=MaximalIncrease(A[l:m])
```

```
(y,y')=MaximalIncrease(A[m+1:r])
z=MinimumSelection(A[l:m])
z'=MaximumSelection(A[m+1:r])
if (A[x']-A[x]>=A[y']-A[y]) AND (A[x']-A[x]>=A[z']-A[z])
  then return (x,x')
if (A[y']-A[y]>=A[x']-A[x]) AND (A[y']-A[y]>=A[z']-A[z])
  then return (y,y')
if (A[z']-A[z]>=A[x']-A[x]) AND (A[z']-A[z]>=A[y']-A[y])
  then return (z,z')
```

Procedure `MinimumSelection(A[l:m])` returns the index of a minimum element in the subarray $A[l:m]$. Procedure `MaximumSelection(A[m+1:r])` returns the index of a maximum element in the subarray $A[m+1:r]$.

The running time of both procedures is proportional to the number of elements in the corresponding subarrays. Thus, the overall running time of the algorithm satisfies the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n/2) + O(n) & \text{if } n > 1. \end{cases}$$

This is essentially the same recurrence as for merge sort, hence the algorithm runs in $T(n) = O(n \log n)$ time.

The crucial part of the algorithm is finding a minimum element in $A[l:m]$ and a maximum element in $A[m+1:r]$. The usual approach is to scan the entire subarrays, which takes time proportional to their sizes. However, it is unnecessary to traverse the subarrays repeatedly, instead, we can compute these values recursively.

Procedure `UpgradedMaximalIncrease(A[l:r])` returns indices $l \leqslant i \leqslant j \leqslant r$ for which $A[j]-A[i]$ is maximal in $A[l:r]$ and indices $l \leqslant \min, \max \leqslant r$ for which $A[\min]$ is a minimum and $A[\max]$ is a maximum element in $A[l:r]$. The procedure must initially be invoked with the parameter $A[1:n]$.

```
UpgradedMaximalIncrease(A[l:r])
if l=r then return (l,l,l,l)
m=[(l+r)/2]
(x,x',l_min,l_max)=UpgradedMaximalIncrease(A[l:m])
(y,y',r_min,r_max)=UpgradedMaximalIncrease(A[m+1:r])
z=l_min
z'=r_max
if (A[x']-A[x]>=A[y']-A[y]) AND (A[x']-A[x]>=A[z']-A[z])
  then return (x,x',min(l_min,r_min),max(l_max,r_max))
if (A[y']-A[y]>=A[x']-A[x]) AND (A[y']-A[y]>=A[z']-A[z])
```

```
  then return (y,y',min(l_min,r_min),max(l_max,r_max))
if (A[z']-A[z]>=A[x']-A[x]) AND (A[z']-A[z]>=A[y']-A[y])
  then return (z,z',min(l_min,r_min),max(l_max,r_max))
```

Now, the overall running time of the upgraded algorithm can be expressed by the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n/2) + O(1) & \text{if } n > 1. \end{cases}$$

Assume, for the sake of simplicity, that $n = 2^k$ is a power of 2. Then, for appropriate positive constants c_1 and c_2, we have

$$\begin{aligned} T(n) &= c_1 + 2T(n/2) \\ &= c_1 + 2(c_1 + 2T(n/2^2)) \\ &= c_1 + 2c_1 + 2^2T(n/2^2) \\ &= c_1 + 2c_1 + 2^2(c_1 + 2T(n/2^3)) \\ &= c_1 + 2c_1 + 2^2c_1 + 2^3T(n/2^3) = \cdots \\ &= c_1 + 2c_1 + 2^2c_1 + \cdots + 2^{k-1}c_1 + 2^kT(n/2^k) \\ &= c_1(1 + 2 + 2^2 + \cdots + 2^{k-1}) + 2^kT(n/2^k) \\ &= c_1(2^k - 1) + 2^kT(n/2^k) \\ &= c_1(n-1) + nT(1) \\ &= c_1(n-1) + c_2n \\ &= (c_1 + c_2)n - c_1. \end{aligned}$$

Thus the upgraded algorithm runs in $T(n) = O(n)$ time.

Master method

The following theorem is occasionally helpful in the analysis of divide and conquer algorithms.

Theorem. Consider the recurrence

$$T(n) = aT(n/b) + f(n)$$

defined on the nonnegative integers, where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is a positive function. (In fact, we should write $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ instead of $T(n/b)$ in the recurrence, but this does not affect the asymptotic result). Then:

- If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

- If $f(n) = \Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and furthermore

$$af(n/b) \leqslant cf(n)$$

for some constant $c < 1$ and all sufficiently large n, then

$$T(n) = \Theta(f(n)).$$

We will not prove this theorem here, as the proof is somewhat technical. Instead, we illustrate its use through several examples.

As a first example, suppose that the following recurrence describes the running time of an algorithm:

$$T(n) = 9T(n/3) + n.$$

Here $a = 9$, $b = 3$ and $f(n) = n$. Since $n^{\log_b a} = n^{\log_3 9} = n^2$, and $f(n) = O(n^{\log_3 9 - \varepsilon})$ with $\varepsilon = 1$, it follows from the first case of the theorem that

$$T(n) = \Theta(n^2).$$

As a second example, consider the recurrence

$$T(n) = T(2n/3) + 1.$$

Here $a = 1$, $b = 3/2$ and $f(n) = 1$. Since $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$, it follows from the second case of the theorem that

$$T(n) = \Theta(\log n).$$

As a third example, consider the recurrence

$$T(n) = 3T(n/4) + n \log n.$$

Here $a = 3$, $b = 4$, and $f(n) = n \log n$. We have $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$.

Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, for some $\varepsilon > 0$, for example $\varepsilon = 0.2$, we can apply the third case of the theorem, provided the "regularity condition" holds. Observe that

$$af(n/b) = 3(n/4)\log(n/4) \leqslant (3/4)n\log n = cf(n),$$

where $c = 3/4 < 1$. Thus

$$T(n) = \Theta(n\log n).$$

As a last example, consider the recurrence

$$T(n) = 2T(n/2) + n\log n.$$

Here $a = 2$, $b = 2$, and $f(n) = n\log n$. We have $n^{\log_b a} = n^{\log_2 2} = n$.

At first glance, this might appear to fall under the third case of the theorem, since $f(n) = n\log n$ grows asymptotically faster than $n^{\log_b a} = n$. However, $n\log n$ is not polynomially larger than n, because

$$\frac{f(n)}{n^{\log_b a}} = \frac{n\log n}{n} = \log n,$$

and $\log n$ grows more slowly than n^{ε} for any positive constant ε.

Therefore, the Master Theorem does not apply directly in this case. Using a more refined analysis, one can show that

$$T(n) = \Theta(n\log^2 n).$$

Karatsuba's fast integer multiplication

The problem we consider is an extremely basic one: the multiplication of two base 2 integers. Students in elementary school are taught a concrete (and quite efficient) algorithm to multiply two n-digit numbers x and y. First, one computes "partial products" by multiplying x by each digit of y separately, then, taking into account the digits' place value, adds up all these partial products. (In elementary school this is usually done in base 10, but it works in exactly the same way in base 2.) Counting a single operation on a pair of bits as one primitive step in this computation, it takes $O(n)$ time to compute each partial product, and $O(n)$ time to add it to the running sum of all partial products so far. Since there are n partial products, this gives an overall running time of $O(n^2)$

Is it possible to improve on $O(n^2)$ time using a different, recursive approach of performing the multiplication? Yes, it is. The improved algorithm,

due to Karatsuba, is based on a more clever way of breaking the product into partial sums.

For the sake of simplicity, assume that n is a power of 2. Start by writing x as $x_1 2^{n/2} + x_0$. In other words, x_1 corresponds to the "high-order" $n/2$ bits, and x_0 corresponds to the "low-order" $n/2$ bits. Similarly, we write $y = y_1 2^{n/2} + y_0$. Thus, we have

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0. \end{aligned}$$

This equation reduces the problem of solving a single n-bit instance (multiplying the two n-bit numbers x and y) to the problem of solving four $n/2$-bit instances (computing the products x_1y_1, x_1y_0, x_0y_1 and x_0y_0). This leads to a first candidate for a divide-and-conquer solution: recursively compute these four $n/2$-bit products, and then combine them using the above equation. The combination step requires a constant number of additions of $O(n)$-bit numbers, and therefore takes $O(n)$ time. Thus, the overall running time $T(n)$ satisfies the recurrence

$$T(n) = 4T(n/2) + O(n).$$

By the Master theorem, the solution to this recurrence is

$$T(n) = O(n^{\log_2 4}) = O(n^2).$$

So, in fact, this four-way divide-and-conquer approach turns out to be just a more complicated way of recovering a quadratic-time algorithm!

It turns out that there is a simple trick that lets us determine all of the terms in the expression

$$x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0$$

using just three recursive calls:

$$\begin{aligned} xy &= (x_1 2^{n/2} + x_0)(y_1 2^{n/2} + y_0) \\ &= x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0 \\ &= x_1 y_1 2^n + ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) 2^{n/2} + x_0 y_0 \end{aligned}$$

This equation reduces the problem of solving a single n-bit instance (multiplying the two n-bit numbers x and y) to the problem of solving three $n/2$-bit instances (computing the products x_1y_1, $(x_1 + x_0)(y_1 + y_0)$ and x_0y_0). We ignore for now the issue that $x_1 + x_0$ and $y_1 + y_0$ may have $n/2 + 1$ bits

rather than exactly $n/2$, since this does not affect the asymptotic running time. Now the overall running time $T(n)$ is bounded by the recurrence

$$T(n) = 3T(n/2) + O(n).$$

By the Master theorem, the solution is

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59}).$$

Strassen's fast matrix multiplication

Matrix multiplication is one of the most fundamental operations in computer science and applied mathematics, and it plays a central role in a wide range of problems, most notably in solving systems of linear equations. Many classical methods for solving systems of linear equations, such as Gaussian elimination and LU decomposition, are built almost entirely from sequences of matrix multiplications and related matrix operations. As a result, improvements in the efficiency of matrix multiplication often translate directly into faster algorithms for solving systems of linear equations and other core problems in computational mathematics.

Consider two $n \times n$ matrices:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix}$$

Then the product $C = AB$ is also an $n \times n$ matrix:

$$C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

where

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \quad (1 \leqslant i, j \leqslant n).$$

Computing C in this way requires n^3 elementary multiplications and $n^2(n-1)$ elementary additions. A natural question arises: is there a more efficient method, perhaps using a divide and conquer approach?

For the sake of simplicity, assume that n is a power of 2. We can then partition the matrices A, B, and C into four $n/2 \times n/2$ submatrices as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

The submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ can then be expressed as:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Each of these formulas involves two $n/2 \times n/2$ matrix multiplications and one $n/2 \times n/2$ matrix addition.

Let $S(n)$ and $M(n)$ denote the number of elementary additions and elementary multiplications, respectively, required to multiply two $n \times n$ matrices. Clearly, $S(1) = 0$ and $M(1) = 1$. Then the following recurrences hold:

$$\begin{aligned} S(n) &= 8S(n/2) + 4(n/2)^2, \\ M(n) &= 8M(n/2). \end{aligned}$$

By the Master theorem, we obtain

$$\begin{aligned} S(n) &= \Theta(n^3), \\ M(n) &= \Theta(n^3). \end{aligned}$$

Strassen showed, however, that the submatrices C_{11}, C_{12}, C_{21}, C_{22} can be computed using only seven $n/2 \times n/2$ matrix multiplications, rather than eight, which leads to a faster asymptotic running time.

First, consider the following seven products:

$$\begin{aligned} P_1 &= A_{11}(B_{12} - B_{22}), \\ P_2 &= (A_{11} + A_{12})B_{22}, \\ P_3 &= (A_{21} + A_{22})B_{11}, \\ P_4 &= A_{22}(B_{21} - B_{11}), \\ P_5 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ P_6 &= (A_{12} - A_{22})(B_{21} + B_{22}), \\ P_7 &= (A_{11} - A_{21})(B_{11} + B_{12}). \end{aligned}$$

Then, the submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ can be expressed as:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6, \\ C_{12} &= P_1 + P_2, \\ C_{21} &= P_3 + P_4, \\ C_{22} &= P_5 + P_1 - P_3 - P_7. \end{aligned}$$

Observe that the standard method requires 8 multiplications and 4 additions, while Strassen's method requires only 7 multiplications and 18 additions (and subtractions). Let $S'(n)$ and $M'(n)$ denote the number of elementary additions and elementary multiplications, respectively, needed to multiply two $n \times n$ matrices by Strassen's method. Clearly, $S'(1) = 0$ and $M'(1) = 1$. Then the following recurrences hold:

$$\begin{aligned} S'(n) &= 7S'(n/2) + 18(n/2)^2, \\ M'(n) &= 7M'(n/2). \end{aligned}$$

By the Master theorem, we obtain

$$\begin{aligned} S'(n) &= \Theta(n^{\log_2 7}) = O(n^{2.81}), \\ M'(n) &= \Theta(n^{\log_2 7}) = O(n^{2.81}). \end{aligned}$$

Finally, note that the condition that n is a power of 2 can be easily removed. If n is not a power of 2, let m be the smallest power of 2 greater than n, and define the $m \times m$ matrices

$$A^* = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & a_{2n} & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{pmatrix}$$

and

$$B^* = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} & 0 & \dots & 0 \\ b_{21} & b_{22} & \dots & b_{2n} & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{pmatrix}.$$

Then the product $C^* = A^*B^*$ is an $m \times m$ matrix of the form:

$$C^* = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} & 0 & \dots & 0 \\ c_{21} & c_{22} & \dots & c_{2n} & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 \end{pmatrix}.$$

Applying Strassen's algorithm to A^* and B^* to compute C^*, and hence C, requires $O(m^{2.81}) = O((2n)^{2.81}) = O(n^{2.81})$ elementary additions and elementary multiplications.

Fast polynomial multiplication

Suppose we are given a polynomial

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{m-1}x^{m-1}$$

of degree $m - 1$ and a polynomial

$$B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1}.$$

of degree $n - 1$, and we wish to compute their product

$$C(x) = A(x)B(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{m+n-2}x^{m+n-2}$$

of degree $m + n - 2$, where

$$c_k = \sum_{\substack{i<m,\, j<n \\ i+j=k}} a_ib_j \quad (0 \leqslant k \leqslant m + n - 2).$$

The straightforward method computes the product a_ib_j for every pair (i, j), requiring $O(mn)$ elementary arithmetic operations. Could we design a more efficient algorithm? Surprisingly, this is possible using a powerful technique known as the Fast Fourier Transform.

Fast polynomial multiplication is not only a fundamental problem in its own right, but also a key building block for fast integer arithmetic. An integer can be represented as a polynomial whose coefficients are blocks of digits (or bits), and multiplying two integers then corresponds to multiplying

these two polynomials and combining the resulting coefficients with carries. Just as improvements in matrix multiplication accelerate the solution of systems of linear equations, advances in fast polynomial multiplication translate into faster algorithms for large-number arithmetic, with broad impact across computational number theory, cryptography, and scientific computing.

For the sake of simplicity, assume that $n = m$ and that this common value is a power of 2. First we evaluate the polynomials $A(x)$ and $B(x)$ at the $(2n)^{\text{th}}$ complex roots of unity $\omega_{j,2n} = e^{2\pi ij/2n}$ for all $0 \leqslant j \leqslant 2n-1$. These are the complex numbers satisfying $(\omega_{j,2n})^{2n} = 1$. Using a divide and conquer algorithm, we can perform all these evaluations efficiently in $O(n \log n)$ time, rather than the naive $O(n^2)$ time required to evaluate each point individually.

One can easily check that a polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

can be written as

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2).$$

where

$$A_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{(n-2)/2}$$

and

$$A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{(n-2)/2}.$$

Then

$$A(\omega_{j,2n}) = A_{\text{even}}(\omega_{j,2n}^2) + \omega_{j,2n} A_{\text{odd}}(\omega_{j,2n}^2).$$

for all $0 \leqslant j \leqslant 2n-1$. Observe that

$$\omega_{j,2n}^2 = (e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$$

which is an nth complex root of unity. Thus, if we recursively evaluate the polynomials $A_{\text{even}}(x)$ and $A_{\text{odd}}(x)$ of degree $(n-2)/2$ at the nth complex roots of unity, then each value $A(\omega_{j,2n})$ can be obtained using a constant number of elementary arithmetic operations, for all $0 \leqslant j \leqslant 2n-1$.

Let $T(n)$ denote the number of elementary arithmetic operations needed to evaluate a polynomial of degree $n-1$ at the $(2n)^{\text{th}}$ complex roots of unity. Then, by the above argument,

$$T(n) = 2T(n/2) + O(n).$$

This recurrence is the same as the one arising in the analysis of merge sort, and therefore $T(n) = O(n \log n)$.

We can evaluate the polynomial $B(x)$ at the $(2n)^{\text{th}}$ complex roots of unity in the same way.

We then evaluate the polynomial $C(x) = A(x)B(x)$ at $\omega_{j,2n}$ for each $0 \leqslant j \leqslant 2n-1$. This requires only $O(n)$ elementary arithmetic operations, using the precomputed values $A(\omega_{j,2n})$ and $B(\omega_{j,2n})$.

Finally, we determine the coefficients of the polynomial $C(x)$ from its values at the $(2n)^{\text{th}}$ complex roots of unity. That is, we seek to recover the coefficients of

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2n-1} x^{2n-1},$$

a polynomial of degree at most $2n-1$, from its values $C(\omega_{s,2n})$ at the $(2n)^{\text{th}}$ complex roots of unity.

Introduce the polynomial

$$D(x) = d_0 + d_1 x + d_2 x^2 + \cdots + d_{2n-1} x^{2n-1}$$

where $d_s = C(\omega_{s,2n})$ for all $0 \leqslant s \leqslant 2n-1$.

The values of the polynomial $D(x)$ at the $(2n)^{\text{th}}$ complex roots of unity can be computed as follows:

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n}) \omega_{j,2n}^s \\ &= \sum_{s=0}^{2n-1} \left(\sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t \right) \omega_{j,2n}^s \\ &= \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} \omega_{s,2n}^t \omega_{j,2n}^s \right) \\ &= \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} e^{2\pi i (st+js)/2n} \right) \\ &= \sum_{t=0}^{2n-1} c_t \left(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s \right). \end{aligned}$$

Observe that if ω is a $(2n)^{\text{th}}$ complex root of unity different from 1, then

$$\sum_{s=0}^{2n-1} \omega^s = 0.$$

Indeed, ω is a root of the polynomial

$$x^{2n} - 1 = (x-1)(x^{2n-1} + x^{2n-2} + \cdots + x^2 + x + 1),$$

and since $\omega \neq 1$, it must be a root of the second factor on the right hand side.

Hence, the only nonzero term in the outer sum occurs when $\omega_{t+j,2n} = 1$, i.e., when $t + j$ is a multiple of $2n$. This can only happen when

$$t = \begin{cases} 0 & \text{if } j = 0, \\ 2n - j & \text{if } 1 \leqslant j \leqslant 2n - 1. \end{cases}$$

For this value

$$\sum_{s=0}^{2n-1} \omega_{t+j,2n}^{s} = \sum_{s=0}^{2n-1} 1 = 2n,$$

and therefore

$$D(\omega_{j,2n}) = \begin{cases} 2nc_0 & \text{if } j = 0, \\ 2nc_{2n-j} & \text{if } 1 \leqslant j \leqslant 2n - 1. \end{cases}$$

The values $D(\omega_{j,2n})$ can be computed using the same divide and conquer algorithm as above using $O(n \log n)$ elementary arithmetic operations. Thus the coefficients of the polynomial $C(x)$ can also be determined from its values at the $(2n)^{\text{th}}$ complex roots of unity in the same asymptotic time.

To sum up, the overall running time of the algorithm is $O(n \log n)$.

Closest pair of points

The problem we consider is very simple to state: Given a set of n points $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ in the plane, find the pair that is closest together (if there are more than one such pair, it is enough to find just one, for the sake of simplicity).

This problem was investigated by M. I. Shamos and D. Hoey in the early 1970s as part of their project to develop efficient algorithms for basic computational primitives in geometry. These algorithms formed the foundations of the then-fledgling field of computational geometry, and they have since found their way into areas such as computer graphics, computer vision, geographic information systems, and molecular modeling. Although the closest-pair problem is one of the most natural algorithmic problems in geometry, it is surprisingly hard to find an efficient algorithm for it. It is immediately clear that there is an $O(n^2)$-time solution — compute the distance between each pair of points and take the minimum — and so Shamos and Hoey asked whether an algorithm asymptotically faster than quadratic could be found. It took quite a long time before they resolved this question, and the $O(n \log n)$-time algorithm we give below is essentially the one they discovered.

The algorithm is based on the divide and conquer principle. The following parameters are passed in each recursive call:

- a subset $\mathcal{Q}$ of the set of points $\mathcal{P}$ (initially $\mathcal{Q} = \mathcal{P}$ itself),
- an array X containing the points in $\mathcal{Q}$, sorted in increasing order of their x-coordinates,
- an array Y containing the points in $\mathcal{Q}$, sorted in increasing order of their y-coordinates.

To avoid sorting the subarrays in each recursive call, we begin by sorting the points of $\mathcal{P}$ in increasing order of their x- and y-coordinates, respectively. In each recursive call, we then pass the appropriate subarrays of these sorted arrays.

A recursive step proceeds as follows. If $|\mathcal{Q}| \leq 3$, we determine the minimal distance by simply comparing the distances between all pairs of points. The really interesting case is when $|\mathcal{Q}| > 3$, in which we take the following steps.

First, we determine a vertical line ℓ that splits $\mathcal{Q}$ into two subsets $\mathcal{Q}_L$ and $\mathcal{Q}_R$, where:

- $|\mathcal{Q}_L| = \lceil \mathcal{Q}/2 \rceil$ and $|\mathcal{Q}_R| = \lfloor \mathcal{Q}/2 \rfloor$,
- each point in $\mathcal{Q}_L$ lies either to the left of ℓ or on ℓ,
- each point in $\mathcal{Q}_R$ lies either to the right of ℓ or on ℓ.

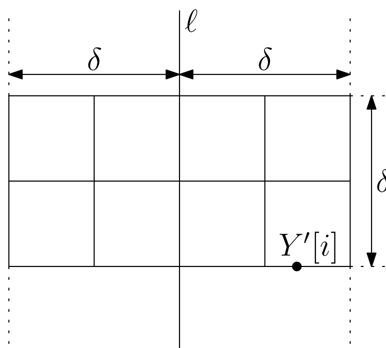
We also split the array X into the arrays X_L and X_R, which consist of points in $\mathcal{Q}_L$ and $\mathcal{Q}_R$, respectively, sorted in increasing order of their x-coordinates. Similarly, we split the array Y into the arrays Y_L and Y_R, which consist of points in $\mathcal{Q}_L$ and $\mathcal{Q}_R$, respectively, sorted in increasing order of their y-coordinates.

Next, the algorithm recursively searches for the closest pairs of points in both sets $\mathcal{Q}_L$ and $\mathcal{Q}_R$. The parameters of the first recursive call are the set $\mathcal{Q}_L$ and the arrays X_L and Y_L, while for the second call they are the set $\mathcal{Q}_R$ and the arrays X_R and Y_R. Let δ_L and δ_R be the minimal distances found in $\mathcal{Q}_L$ and $\mathcal{Q}_R$, respectively, and let $\delta = \min(\delta_L, \delta_R)$. Now, the closest pair of points is

- either a pair with distance δ, found by one of the recursive calls,
- or a pair such that one point lies in $\mathcal{Q}_L$, and the other lies in $\mathcal{Q}_R$.

The crucial step of the algorithm is to determine whether there is a pair of the latter type whose distance is smaller than δ. Observe that if such a pair exists, then both points must lie within distance δ of the line ℓ. We therefore create a new array Y' from the array Y, which contains exactly the points of $\mathcal{Q}$ whose distance from ℓ is at most δ. The elements of Y' are, of course, still in increasing order of their y-coordinates.

Next, for each point $p \in Y'$, we look for all points in Y' that are at distance at most δ from p and lie either on the horizontal line through p or above it. It is easy to see that there can be at most 7 such points in $Y' \setminus \{p\}$. Indeed, if $i < j$ and the distance between $Y'[i]$ and $Y'[j]$ is at most δ, then these two points lie in a rectangle of size $\delta \times 2\delta$, as shown below.



Assigning the points on ℓ that belong to $\mathcal{Q}_L$ to the left small squares, and the points on ℓ that belong to $\mathcal{Q}_R$ to the right small squares, none of the eight small squares can contain more than one point, since the greatest possible distance within such a square is $\delta\sqrt{2}/2 < \delta$.

We now proceed as follows. For each point $p \in Y'$, we compute the distances from p to the next 7 points in Y'. Let δ' denote the overall minimum of these distances. Evidently, if $\delta' < \delta$, then the minimal distance in $\mathcal{Q}$ is δ', otherwise, it is δ.

Let $T(n)$ be the running time of the algorithm on an n element point set, excluding the initial sortings. The recursive calls take $2T(n/2)$ time. Splitting the point set, computing the sorted arrays X_L, X_R, Y_L, Y_R and Y', and computing δ' is performed in $O(n)$ time. Thus, we obtain the following recurrence:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 3, \\ 2T(n/2) + O(n) & \text{if } n > 3. \end{cases}$$

This is essentially the same recurrence we encountered for merge sort, and hence

$$T(n) = O(n \log n).$$

Since the initial sortings also take $O(n \log n)$ time, e.g., by merge sort, the overall running time of the algorithm is $O(n \log n)$.

Dynamic programming

We now turn to a more powerful and subtle design technique: dynamic programming. Like the divide-and-conquer method, dynamic programming solves problems by combining the solutions to subproblems. Divide-and-conquer algorithms partition the problem into independent subproblems, solve these subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when they share common subsubproblems. In such cases, a divide-and-conquer algorithm does more work than necessary by repeatedly solving the same subsubproblems. A dynamic-programming algorithm solves each subsubproblem only once and stores its result in a table, avoiding recomputation whenever the same subsubproblem is encountered again. We will develop a dynamic programming algorithm in two stages: first as a recursive procedure, and then by reinterpreting this procedure, as an iterative algorithm that builds up solutions to progressively larger subproblems.

Matrix-chain multiplication

We are given a sequence of n matrices $A_1, A_2, \ldots, A_n$, and we wish to compute the product $A_1 A_2 \cdots A_n$.

We can evaluate the expression using the standard algorithm for multiplying pairs of matrices once we have chosen a parenthesization. Since matrix multiplication is associative, the result is the same regardless of how we parenthesize the product. However, although the result is always the same, the choice of parenthesization can have a significant impact on the cost of evaluating the product.

First, consider the cost of multiplying two matrices. We can multiply matrices A and B only if their dimensions are compatible, which means that the number of columns in A equals the number of rows in B.

```
MatrixMultiplication(A,B)
if column[A]<>row[B]
  then
    error 'incompatible dimensions'
  else
    for i=1 to row[A] do
      for j=1 to column[B] do
        C[i,j]=0
        for k=1 to column[A] do
          C[i,j]=C[i,j]+A[i,k]B[k,j]
    return C
```

Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix. Then the product $C = AB$ is a $p \times r$ matrix. Computing C requires pqr scalar multiplications. In what follows, we measure the cost of matrix multiplication by the number of scalar multiplications.

To illustrate the effect of parenthesization on the computational cost, consider the following example. Let A_1 be a 10×100 matrix, let A_2 be a 100×5 matrix, and let A_3 be a 5×50 matrix. We can parenthesize $A_1A_2A_3$ in two different ways:

$$(A_1A_2)A_3,$$
$$A_1(A_2A_3).$$

In the first case, computing A_1A_2, which is a 10×5 matrix, requires

$$10 \cdot 100 \cdot 5 = 5000$$

scalar multiplications. Multiplying this result by A_3 then requires an additional

$$10 \cdot 5 \cdot 50 = 2500$$

scalar multiplications. Altogether, this amounts to 7500 scalar multiplications. In the second case, computing A_2A_3 requires

$$100 \cdot 5 \cdot 50 = 25000$$

scalar multiplications and produces a 100×50 matrix. Multiplying this result on the left by A_1 then requires an additional

$$10 \cdot 100 \cdot 50 = 50000$$

scalar multiplications. Altogether, this amounts to 75000 scalar multiplications, which far exceeds the number required in the first case.

The matrix-chain multiplication problem can be formalized as follows. We are given a sequence of n matrices $A_1, A_2, \ldots, A_n$, where, for $i = 1, 2, \ldots, n$, the matrix A_i has dimensions $p_{i-1} \times p_i$. The goal is to find a parenthesization of the product

$$A_1 A_2 \cdots A_n$$

that minimizes the number of scalar multiplications required to compute it. Note that the problem concerns finding the optimal parenthesization, not actually performing the matrix multiplications. In some cases, computing the optimal parenthesization may take more time than the savings gained by applying it.

We show that testing every possible parenthesization is not effective. Let $P(n)$ denote the number of possible parenthesizations of the product of n matrices. For $n = 1$ we have only one matrix, so $P(n) = 1$. For $n > 1$, the product can be written as the multiplication of two parenthesized subproducts. Since the numbers of parenthesizations of these products are independent, we obtain the following recurrence:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n > 1. \end{cases}$$

This recurrence is well known: its solution is the sequence of Catalan numbers, which grow asymptotically as $\Omega(4^n/n^{3/2})$. Thus, the number of possible parenthesizations is exponential in n.

We now show how to solve the problem efficiently using dynamic programming. The first step in such a solution is to define appropriate subproblems. Here, the subproblems are to optimally parenthesize the subproducts $A_i A_{i+1} \cdots A_j$ for $1 \leqslant i \leqslant j \leqslant n$. In the non-trivial case, when $i < j$, any parenthesization splits the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1} for some $i \leqslant k < j$. Introducing the notation $A_{i..j}$ for the product $A_i A_{i+1} \cdots A_j$, we see that $A_{i..j}$ can be obtained by first computing $A_{i..k}$ and $A_{k+1..j}$, and then multiplying the results. Thus, the cost of this parenthesization is the cost of computing $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying these two results. The following observation is essential and is known as the "optimal substructure property".

Proposition. Suppose the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesizations of the subproducts $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$ are also optimal.

Indeed, if there were a lower-cost parenthesization of, say, $A_i A_{i+1} \cdots A_k$,

then substituting it into the optimal parenthesization of $A_iA_{i+1}\cdots A_j$ would yield a lower-cost parenthesization of $A_iA_{i+1}\cdots A_j$, a contradiction.

This means that we can build the optimal parenthesization of $A_iA_{i+1}\cdots A_j$ by splitting the problem into two subproblems, finding the optimal solutions to these subproblems, and then combining the solutions. It is important to consider every possible split when determining the optimal place to divide the product. Ignoring some possibilities may lead to a suboptimal solution.

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. Let $m[i,j]$ denote the minimum number of scalar multiplications needed to compute the product $A_{i..j}$. If $i = j$, then $A_{i..i} = A_i$. No multiplications are needed in this case, so $m[i,i] = 0$, for all $i = 1, 2, \ldots, n$. If $i < j$, consider the optimal parenthesization of $A_iA_{i+1}\cdots A_j$. Suppose it splits the product between A_k and A_{k+1} for some $i \leqslant k$. Then, by the optimal substructure property,

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Note that $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is a $p_k \times p_j$ matrix.

The previous equation assumes that the splitting index k is known, which is not the case. However, the number of possible values for k is $j-i$, namely, $k = i, i+1, \ldots, j-1$. Since one of these yields the optimal parenthesization, we check all of them to find the minimum:

$$m[i,j] = \begin{cases} 0 & \text{when } i = j, \\ \min\limits_{i \leqslant k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & \text{when } i < j. \end{cases}$$

The cost of the optimal solution can be computed bottom-up by storing the values of the recursive formula in a table $m[1:n, 1:n]$. The input to the algorithm is the array of dimensions $p = [p_0, p_1, \ldots, p_n]$.

```
MatrixChainMultiplication(p)
for i=1 to n do
   m[i,i]=0
for l=2 to n do
   for i=1 to n-l+1 do
       j=i+l-1
       m[i,j]=INFINITY
       for k=i to j-1 do
          q=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j
          if q<m[i,j] then m[i,j]=q
return m
```

The first two lines of the algorithm set $m[i,i] = 0$. Then it computes the values $m[i, i+1]$, i.e., the minimum costs for products of length $l = 2$, in the first iteration of the loop at line 3. The second iteration computes the values $m[i, i+2]$, i.e., the minimum costs for products of length $l = 3$, and so on in the subsequent iterations. Note that when computing $m[i, j]$, the values $m[i, k]$ and $m[k+1, j]$ have already been computed. Since $m[i, j]$ is defined only for $i \leqslant j$, only the portion of the table m above the main diagonal is used.

The three nested loops of `MatrixChainMultiplication(p)` take $O(n^3)$ time, as all the loop variables l, i and k are bounded by n.

To reconstruct an optimal parenthesization, we introduce an additional table. For each $1 \leqslant i < j \leqslant n$, let $s[i, j]$ be the index k where the optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$, i.e., the index for which

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j.$$

The table $s[1:n, 1:n]$ can be computed easily alongside the previous procedure.

```
MatrixChainMultiplication(p)
for i=1 to n do
   m[i,i]=0
for l=2 to n do
   for i=1 to n-l+1 do
      j=i+l-1
      m[i,j]=INFINITY
      for k=i to j-1 do
         q=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j
         if q<m[i,j] then
            m[i,j]=q
            s[i,j]=k
return m,s
```

Using the table $s[1:n, 1:n]$, it becomes fairly easy to reconstruct an optimal parenthesization. Now $s[1, n]$ stores the index k where the optimal parenthesization splits the product $A_1 A_2 \cdots A_n$ into two parts, i.e., between A_k and A_{k+1}. Thus, the last multiplication in the optimal computation of $A_{1..n}$ is

$$A_{1..s[1,n]} A_{s[1,n]+1..n}.$$

The optimal parenthesizations for $A_{1..s[1,n]}$ and $A_{s[1,n]+1..n}$ can be computed recursively:

- $s[1, s[1, n]]$ gives the last multiplication in the optimal computation of $A_{1..s[1,n]}$,
- $s[s[1, n] + 1, n]$ gives the last multiplication in the optimal computation of $A_{s[1,n]+1..n}$.

The following recursive procedure prints an optimal parenthesization. It must initially be invoked with the parameters $(s, 1, n)$.

```
OptimalParenthesization(s,i,j)
if j=i
   then
      print 'A'i
   else
      print '('
      OptimalParenthesization(s,i,s[i,j])
      OptimalParenthesization(s,s[i,j]+1,j)
      print ')'
```

Sequence Alignment

Dictionaries on the Web seem to get more and more useful: often it seems easier to pull up a bookmarked online dictionary than to get a physical dictionary down from the bookshelf. Many online dictionaries also offer functions that you can't get from a printed one: if you're looking for a definition and type in a word it doesn't contain — say, ocurrance — it will come back and ask, "Perhaps you mean occurrence?" How does it do this? To decide what you probably meant, it is natural to search the dictionary for the word most "similar" to the one you typed. To do this, we must answer the question: how should we define similarity between two words or strings? In the early 1970s, the molecular biologists Needleman and Wunsch proposed a definition of similarity that remains, largely unchanged, the standard framework for sequence comparison today.

We are given the character sequences $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$. Consider the sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ which represent the positions in X and Y, respectively, and let M be a matching between them. Such a matching M is called a sequence alignment if, for all (i, j), $(i', j') \in M$, whenever $i < i'$, we have $j < j'$.

There is a natural visualization of sequence alignments. Consider, for example, the sequences

$$(G, A, T, C, G, G, C, A, T) \quad \text{and} \quad (C, A, A, T, G, T, G, A, A, T, C),$$

and the alignment

$$\{(1,1),(2,3),(3,4),(5,5),(6,7),(7,8),(8,9),(9,10)\}.$$

We can draw this alignment in the following way:

```
G - A T C G - G C A T -
C A A T - G T G A A T C
```

We can interpret this picture as a way of transforming the first sequence into the second one as follows. Replace the character G at the first position with C. Then, after the first position, insert the character A. Next, delete the C in the fourth position. After the fifth position, insert the character T. Then, replace the C in the seventh position with A. Finally, after the ninth position, insert the character C.

The definition of similarity is based on finding the optimal alignment between X and Y according to the following criteria. Suppose M is a given alignment between X and Y.

- There is a parameter $g > 0$ that defines a gap penalty. For each position of X or Y that is not matched in M — that is, a gap — we incur a cost of g.
- For each pair (P, Q) of letters in our alphabet, there is a mismatch cost $c[P, Q] \geqslant 0$ for lining up P with Q. Thus, for each $(i, j) \in M$, we pay the appropriate mismatch cost $c[x_i, y_j]$ for lining up x_i with y_j. One generally assumes that $c[P, P] = 0$ for each letter P — there is no mismatch cost to line up a letter with another copy of itself.

The cost of M is the sum of its gap and mismatch costs, and we seek an alignment of minimum cost. The minimum cost indicates the similarity between the sequences X and Y, with lower values corresponding to greater similarity.

We solve the problem by dynamic programming. To define appropriate subproblems, we introduce the concept of a prefix. The subsequence $X_i = (x_1, x_2, \ldots, x_i)$ is called the ith prefix of the sequence $X = (x_1, x_2, \ldots, x_m)$ for $i = 0, 1, \ldots, m$. For example, if $X = (G, A, T, C, G, G, C)$ then $X_4 = (G, A, T, C)$ and X_0 is the empty sequence.

First, we show that the sequence alignment problem has an optimal substructure property. Observe that if M is a sequence alignment between X and Y and $(m, n) \notin M$, then either the mth position of X or the nth position of Y is not present in the pairs of M. Indeed, assume, for contradiction, that $(m, n) \notin M$ and there exist positive integers $i < m$ and $j < n$ such that

$(m, j) \in M$ and $(i, n) \in M$. However, this contradicts the definition of a sequence alignment because $(i, n), (m, j) \in M$ with $i < m$ but $n > j$.

Here is an equivalent way to write this.

Proposition. Let M be an optimal sequence alignment between the sequences X and Y. Then one of the followings holds:

(1) $(m, n) \in M$,

(2) the mth position of X is not present in the pairs of M,

(3) the nth position of Y is not present in the pairs of M.

And here is the optimal substructure property.

Proposition. Let M be an optimal sequence alignment between the sequences X and Y.

(1) If $(m, n) \in M$, then $M \setminus \{(m, n)\}$ is an optimal sequence alignment between X_{m-1} and Y_{n-1}.

(2) If the mth position of X is not present in the pairs of M, then M is an optimal sequence alignment between X_{m-1} and Y.

(3) If the nth position of Y is not present in the pairs of M, then M is an optimal sequence alignment between X and Y_{n-1}.

Proof.

(1) If there were a lower-cost sequence alignment between X_{m-1} and Y_{n-1} than $M \setminus \{(m, n)\}$, then appending (m, n) to this alignment would yield a lower-cost sequence alignment between X and Y than M, contradicting the optimality of M.

(2) If there were a lower-cost sequence alignment between X_{m-1} and Y than M, then this alignment would also be a lower-cost sequence alignment between X and Y than M, contradicting the optimality of M.

(3) If there were a lower-cost sequence alignment between X and Y_{n-1} than M, then this alignment would also be a lower-cost sequence alignment between X and Y than M, contradicting the optimality of M.

What does this mean?

- If an optimal sequence alignment between X and Y contains (m, n), then the cost of this sequence alignment is $c[x_m, y_n]$ plus the cost of an optimal sequence alignment between X_{m-1} and Y_{n-1}.
- If an optimal sequence alignment between X and Y does not contain the mth position of X in its pairs, then the cost of this optimal sequence alignment is g plus the cost of an optimal sequence alignment between X_{m-1} and Y.
- If an optimal sequence alignment between X and Y does not contain the nth position of Y in its pairs, then the cost of this optimal sequence alignment is g plus the cost of an optimal sequence alignment between X and Y_{n-1}.

In computing the optimal sequence alignment, we consider all three cases and select the one with minimal cost.

For each $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$, let $a[i, j]$ denote the cost of an optimal sequence alignment between X_i and Y_j. It's easy to see that $a[i, 0] = ig$ and $a[0, j] = jg$ for all $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$. Otherwise, we use the above observation. Thus, the recursive formula for $a[i, j]$ is

$$a[i, j] = \min\{c[x_i, y_j] + a[i-1, j-1], g + a[i-1, j], g + a[i, j-1]\}.$$

We fill the table $a[0 : m, 0 : n]$, which stores the values of the recursive formula, in row-major order: first the first row from left to right, then the second row, and so on. The cost of the optimal sequence alignment is contained in $a[m, n]$.

```
SequenceAlignment(X,Y)
for j=0 to n do
  a[0,j]=j*g
for i=1 to m do
  a[i,0]=i*g
  for j=1 to n do
    a[i,j]=c[x_i,y_j]+a[i-1,j-1]
    if a[i,j]>g+a[i-1,j] then
      a[i,j]=g+a[i-1,j]
    if a[i,j]>g+a[i,j-1] then
      a[i,j]=g+a[i,j-1]
return a
```

The procedure runs in $O(mn)$ time. To recover an optimal solution we introduce an additional table. For each $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$ let

$b[i,j]$ point to the entry in the table $a[0:m,0:n]$ that corresponds to the choice yielding the optimal value in the computation of $a[i,j]$. The table $b[1:m,1:n]$ can be computed easily alongside the previous procedure.

```
SequenceAlignment(X,Y)
for j=0 to n do
  a[0,j]=j*g
for i=1 to m do
  a[i,0]=i*g
  for j=1 to n do
    a[i,j]=c[x_i,y_j]+a[i-1,j-1]
    b[i,j]=(i-1,j-1)
    if a[i,j]>g+a[i-1,j] then
      a[i,j]=g+a[i-1,j]
      b[i,j]=(i-1,j)
    if a[i,j]>g+a[i,j-1] then
      a[i,j]=g+a[i,j-1]
      b[i,j]=(i-1,j-1)
return a,b
```

Using the table $b[1:m,1:n]$, it is easy to recover an optimal sequence alignment: begin at $b[m,n]$ and trace back through the table by following the pointers. Moving from $b[i,j]$ to $b[i-1,j-1]$ means that the pair (i,j) is part of the optimal sequence alignment. The following recursive procedure prints the pairs of positions of an optimal sequence alignment. It must initially be invoked with the parameters (b,m,n).

```
PrintResult(b,i,j)
if i=0 or j=0 then return
if b[i,j]=(i-1,j-1)
  then
    PrintResult(b,i-1,j-1)
    print '(' i ',' j ')'
  else
    if b[i,j]=(i-1,j)
      then PrintResult(b,i-1,j)
      else PrintResult(b,i,j-1)
```

The procedure runs in $O(m+n)$ time, since at least one of i and j decreases at each stage of the recursion.

Longest common subsequence

A sequence $Z = (z_1, z_2, \dots, z_k)$ is a subsequence of another sequence $X = (x_1, x_2, \dots, x_m)$ if there exist indices $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m$ such that $z_j = x_{i_j}$ for each $1 \leqslant j \leqslant k$. For example, $Z = (B, C, D, B)$ is a subsequence of $X = (A, B, C, B, D, A, B)$, the corresponding index sequence is $(2, 3, 5, 7)$. We say that a sequence Z is a common subsequence of the sequences X and Y if Z is a subsequence of both X and Y. For example, if $X = (A, B, C, B, D, A, B)$ and $Y = (B, D, C, A, B, A)$, then (B, C, A) is a common subsequence of them. Observe that there is a longer common subsequence of these sequences as well, e.g., (B, C, B, A). The longest common subsequence problem is the following: Given two sequences $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$, find a common subsequence of maximum length. The problem is closely related to the sequence alignment problem.

Since X has 2^m subsequences, it is not efficient to check each subsequence to see whether it is also a subsequence of Y and then select the longest one among them.

We can solve the problem more efficiently using dynamic programming. To define appropriate subproblems, we recall the notion of a prefix: $X_i = (x_1, x_2, \dots, x_i)$ is called the ith prefix of $X = (x_1, x_2, \dots, x_m)$ for $0 \leqslant i \leqslant m$. First, we show that the longest common subsequence problem has an optimal substructure property.

Proposition. Let $X = (x_1, x_2, \dots, x_m)$ and $Y = (y_1, y_2, \dots, y_n)$ be two sequences, and let $Z = (z_1, z_2, \dots, z_k)$ be a longest common subsequence of them.

(1) If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is a longest common subsequence of X_{m-1} and Y_{n-1}.

(2) If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is a longest common subsequence of X_{m-1} and Y.

(3) If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is a longest common subsequence of X and Y_{n-1}.

Proof.

(1) If $z_k \neq x_m$, then appending $x_m = y_n$ to Z would yield a common subsequence of X and Y of length $k + 1$, a contradiction. Thus, we must have $z_k = x_m = y_n$.

It follows that Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}. If there were a common subsequence of X_{m-1} and Y_{n-1} longer than Z_{k-1}, then appending z_k to it would produce a common subsequence of X and Y longer than Z, which is impossible. Therefore, Z_{k-1} is a longest common subsequence of X_{m-1} and Y_{n-1}.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence of X_{m-1} and Y longer than Z, then it would also be a common subsequence of X and Y longer than Z, a contradiction.

(3) This case is symmetric to the previous one.

By the previous proposition, determining a longest common subsequence of $X = (x_1, x_2, \ldots, x_m)$ and $Y = (y_1, y_2, \ldots, y_n)$ involves either one or two subproblems, depending on whether the last elements match:

- If $x_m = y_n$, then it suffices to find a longest common subsequence of X_{m-1} and Y_{n-1}, and then appending $x_m = y_n$ yields a longest common subsequence of X and Y.
- If $x_m \neq y_n$, then we must solve two subproblems: finding a longest common subsequence of X_{m-1} and Y, and finding a longest common subsequence of X and Y_{n-1}. The longer of the two is a longest common subsequence of X and Y.

For each $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant n$, let $a[i,j]$ denote the length of a longest common subsequence of X_i and Y_j. If $i = 0$ or $j = 0$, i.e., at least one of the sequences has length zero, then the length of a longest common subsequence is zero. Otherwise, we use the above observation. Thus the recursive formula for $a[i,j]$ is

$$a[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ a[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(a[i-1, j], a[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

We fill the table $a[0:m, 0:n]$, which stores the values of the recursive formula, in row-major order: first the first row from left to right, then the second row, and so on. The length of a longest common subsequence is contained in $a[m,n]$.

```
LongestCommonSubsequence(X,Y)
for j=0 to n do
   a[0,j]=0
for i=1 to m do
```

```
   a[i,0]=0
   for j=1 to n do
      if xi=yj
         then
            a[i,j]=a[i-1,j-1]+1
         else
            if a[i-1,j]>=a[i,j-1]
               then a[i,j]=a[i-1,j]
               else a[i,j]=a[i,j-1]
return a
```

The procedure runs in $O(mn)$ time. To recover an optimal solution we introduce an additional table. For each $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$, let $b[i,j]$ point to the entry in the table $a[0 : m, 0 : n]$ that corresponds to the choice yielding the optimal value in the computation of $a[i,j]$. The table $b[1 : m, 1 : n]$ can be computed easily alongside the previous procedure.

```
LongestCommonSubsequence(X,Y)
for j=0 to n do
   a[0,j]=0
for i=1 to m do
   a[i,0]=0
   for j=1 to n do
      if xi=yj
         then
            a[i,j]=a[i-1,j-1]+1
            b[i,j]=(i-1,j-1)
         else
            if a[i-1,j]>=a[i,j-1]
               then
                  a[i,j]=a[i-1,j]
                  b[i,j]=(i-1,j)
               else
                  a[i,j]=a[i,j-1]
                  b[i,j]=(i,j-1)
return a,b
```

Using the table $b[1 : m, 1 : n]$, it is easy to recover a longest common subsequence: begin at $b[m,n]$ and trace back through the table by following the pointers. Moving from $b[i,j]$ to $b[i-1, j-1]$ means that the element $x_i = y_j$ belongs to the longest common subsequence. This method prints the

elements of a longest common subsequence in reverse order. The following recursive procedure prints the elements of a longest common subsequence in the correct order. It must initially be invoked with the parameters (b, X, m, n).

```
PrintLCS(b,X,i,j)
if i=0 or j=0 then return
if b[i,j]=(i-1,j-1)
   then
      PrintLCS(b,X,i-1,j-1)
      print x_i
   else
      if b[i,j]=(i-1,j)
         then PrintLCS(b,X,i-1,j)
         else PrintLCS(b,X,i,j-1)
```

The procedure runs in $O(m + n)$ time, since at least one of i and j decreases at each stage of the recursion.

Knapsack problem

Suppose a hiker is about to go on a trek through a rainforest carrying a single knapsack. Further, suppose that she knows the maximum total weight W that she can carry, and that she has a set of n different useful items that she could potentially take with her, such as a folding chair, a tent, and so on. Assume that each item i has an integer weight w_i and a benefit value v_i representing the utility that the hiker assigns to item i. Her problem, of course, is to maximize the total value of the set T of items she takes with her, without exceeding the weight limit W. That is, her objective is

$$\text{maximize} \quad \sum_{i \in T} v_i \quad \text{subject to} \quad \sum_{i \in T} w_i \leqslant W.$$

We could solve the knapsack problem in $O(2^n)$ time by enumerating all subsets of the items and selecting the one with the highest total benefit among those whose total weight does not exceed W. However, this is highly inefficient. Fortunately, we can derive dynamic programming algorithms that are much faster in most cases.

More formally, we are given a set of n items $t_1, t_2, \ldots, t_n$. Each item t_i has a weight w_i and a value v_i, where w_i and v_i are positive integers. We are also given a knapsack of capacity W, which is also a positive integer. Our task is to select a subset of the items such that the sum of their values is

maximized, while the sum of their weights does not exceed the capacity W of the knapsack.

We present a dynamic programming algorithm for solving the knapsack problem. The subproblems are defined as follows. For each $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant W$ find a subset of items $t_1, t_2, \ldots, t_i$ whose total weight is at most j, and whose total value is as large as possible. Let $a[i,j]$ denote the total value in the optimal solution to this subproblem. Define the base cases by setting $a[i,0] = 0$ for all $i = 0, 1, \ldots, n$ and $a[0,j] = 0$ for all $j = 0, 1, \ldots, W$.

Now let $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant W$, and let $\mathcal{O}$ be an optimal solution to the subproblem where the weight bound is j and the available items are $t_1, t_2, \ldots, t_i$.

If $w_i > j$, then t_i cannot belong to $\mathcal{O}$, and $\mathcal{O}$ is also an optimal solution to the subproblem where the total weight bound is j and the available items are $t_1, t_2, \ldots, t_{i-1}$. Hence $a[i,j] = a[i-1,j]$ in this case.

If $w_i \leqslant j$, then there are two cases.

- If t_i belongs to $\mathcal{O}$, then $\mathcal{O}' = \mathcal{O} \setminus \{t_i\}$ is an optimal solution to the subproblem, where the weight bound is $j - w_i$ and the available items are $t_1, t_2, \ldots, t_{i-1}$. Indeed, if there were a solution to this subproblem of total value greater than that of $\mathcal{O}'$, then adding t_i would yield a solution of total value greater than that of $\mathcal{O}$, a contradiction. In this case $a[i,j] = v_i + a[i-1, j-w_i]$.
- On the other hand, if t_i doesn't belong to $\mathcal{O}$, then $\mathcal{O}$ is obviously an optimal solution to the subproblem, where the weight bound is j and the available items are $t_1, t_2, \ldots, t_{i-1}$. In this case $a[i,j] = a[i-1,j]$

Since we do not know whether t_i belongs to $\mathcal{O}$, we consider both possibilities and choose the one that gives the larger value. Hence

$$a[i,j] = \max(v_i + a[i-1, j-w_i], a[i-1,j])$$

in this case.

To summarize, the recurrence is

$$a[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ a[i-1,j] & \text{if } w_i > j, \\ \max(v_i + a[i-1, j-w_i], a[i-1,j]) & \text{if } i > 0 \text{ and } j \geqslant w_i. \end{cases}$$

We fill the table $a[0:n, 0:W]$, which stores the values of the recursive formula, in row-major order: first the first row from left to right, then the second row, and so on. The value of the optimal solution to the original problem is contained in $a[n,W]$.

```
Knapsack(n,v,w,W)
for j=0 to W do
  a[0,j]=0
for i=1 to n do
  a[i,0]=0
  for j=1 to W do
    if w[i]>j
      then
        a[i,j]=a[i-1,j]
      else
        if v[i]+a[i-1,j-w[i]]>a[i-1,j]
          then a[i,j]=v[i]+a[i-1,j-w[i]]
          else a[i,j]=a[i-1,j]
return a
```

Since each value $a[i,j]$ can be computed in constant time, the algorithm runs in $O(nW)$ time. Note, however, that `Knapsack` is not a polynomial time algorithm in terms of the input size, since the input W can be represented with only $O(\log W)$ bits, while the algorithm's running time depends linearly on W, which is exponential in the length of its binary representation. In fact, the knapsack problem is known to be NP-hard, meaning that no polynomial-time algorithm is known (or likely exists) for the general case.

To recover an optimal set of items, we introduce an additional table. For each $1 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant W$, let $b[i,j]$ be `TRUE` if, in the computation of $a[i,j]$, including item t_i yields a strictly greater total value; otherwise, let $b[i,j]$ be `FALSE`. The table $b[1:n, 0:W]$ can be filled alongside the table $a[0:n, 0:W]$.

```
Knapsack(n,v,w,W)
for j=0 to W do
  a[0,j]=0
for i=1 to n do
  a[i,0]=0
  b[i,0]=FALSE
  for j=1 to W do
    if w[i]>j
      then
        a[i,j]=a[i-1,j]
        b[i,j]=FALSE
      else
        if v[i]+a[i-1,j-w[i]]>a[i-1,j]
```

```
          then
            a[i,j]=v[i]+a[i-1,j-w[i]]
            b[i,j]=TRUE
          else
            a[i,j]=a[i-1,j]
            b[i,j]=FALSE
return a,b
```

Using the table $b[1:n, 0:W]$, we can easily recover an optimal set of items by backtracking from $b[n, W]$.

```
PrintItems
j=W
for i=n downto 1 do
  if b[i,j]=TRUE then
    print i '. item'
    j=j-w[i]
```

Change-making problem

In the change-making problem, we are given a set of coins and we wish to determine, for a given amount N, the minimum number of coins needed to pay N. For instance, given coins of denominations $1, 5, 10, 25$, the minimal representation of $N = 17$ requires 4 coins $(10 + 5 + 1 + 1)$.

Let the coin denominations be $d_1, d_2, \ldots, d_k$. Assume that an arbitrarily large number of coins of each denomination is available and that one of the coins has denomination 1; hence, it is possible to make change for any amount $N \geqslant 1$.

In some cases, the solution can be found using a greedy strategy that selects as many coins of the highest denomination as possible, then as many of the second-highest denomination as possible, and so on. This greedy solution is optimal for the set of coins given above, as we will prove shortly, but fails to be optimal in general. For example, if the available coin denominations are $1, 5, 9, 16$, then the amount 18 would be paid greedily as $16 + 1 + 1$, whereas the optimal solution, $9 + 9$, uses only two coins.

We prove that the above greedy algorithm solves the change-making problem for coin denominations $1, 5, 10, 25$. First, we show that the problem has an optimal substructure property. Suppose we have an optimal solution for making change for value N, and that this optimal solution uses a coin of denomination d_j. Let this solution use c coins in total. We claim that this solution must contain within it an optimal solution to the subproblem of

making change for value $N - d_j$. Indeed, if there were a solution for value $N - d_j$ using fewer than $c - 1$ coins, adding a coin of denomination d_j would yield a solution for value N using fewer than c coins, which contradicts the optimality of the original solution.

Next, we prove that there is always an optimal solution that contains the greedy choice. Specifically, if N is the amount to be payed and d_j is the largest denomination less than or equal to N, then there exists an optimal solution for making change for value N that contains a coin of denomination d_j. Consider an optimal solution to the problem. If the solution already contains a coin of denomination d_j, then we are done. Assume, for contradiction, that it does not. We now distinguish four cases.

(1) If $1 \leqslant N < 5$, then $d_j = 1$. According to our condition, the optimal solution does not contain a coin of denomination 1, which is clearly impossible.

(2) If $5 \leqslant N < 10$, then $d_j = 5$. According to our assumption, the optimal solution does not contain a coin of denomination 5, so it consists only of coins of denomination 1. Replacing five of these coins with one coin of denomination 5 would yield a solution using fewer coins, a contradiction.

(3) If $10 \leqslant N < 25$, then $d_j = 10$. According to our assumption, the optimal solution does not contain a coin of denomination 10, so it consists only of coins of denominations 1 and 5. Some subset of these coins must sum to 10; replacing them with one coin of denomination 10 would yield a solution using fewer coins, a contradiction.

(4) If $N \geqslant 25$, then $d_j = 25$. According to our assumption, the optimal solution does not contain a coin of denomination 25. If it contains three coins of denomination 10, then replacing them with one coin of denomination 25 and one coin of denomination 5 would yield a solution using fewer coins, a contradiction. On the other hand, if it contains at most two coins of denomination 10, then some subset of the coins of denominations 1, 5 and 10 must sum to 25. Replacing these with one coin of denomination 25 would again yield a solution using fewer coins, a contradiction.

This completes the proof of the correctness of the greedy algorithm. The algorithm runs in $O(1)$ time.

As we have already pointed out, the greedy solution fails to be optimal in general. However, the following dynamic programming algorithm always works. For each $0 \leqslant i \leqslant N$, let $a[i]$ denote the minimum number of coins in an optimal solution for making change for value i. Clearly, $a[0] = 0$. If $1 \leqslant i \leqslant N$ and an optimal solution for value i contains a coin with

denomination d_j, then, by the optimal substructure property,

$$a[i] = 1 + a[i - d_j].$$

This recursive equation assumes that the index j is known, which is not the case. Nevertheless, since there are only k possible denominations, we can simply check all of them to find the best choice:

$$a[i] = 1 + \min\{a[i - d_j] \mid 1 \leqslant j \leqslant k,\ d_j \leqslant i\}.$$

We can compute the values $a[i]$ in increasing order of i using a table. The procedure also constructs a table $b[1 : N]$, where $b[i]$ is the denomination of a coin used in an optimal solution for making change for value i.

```
ChangeMaking(N,d)
a[0]=0
for i=1 to N do
  a[i]=INFINITY
  for j=1 to k do
    if i>=d[j] AND 1+a[i-d[j]]<a[i] then
      a[i]=1+a[i-d[j]]
      b[i]=d[j]
return a,b
```

Procedure `ChangeMaking` runs in $O(Nk)$ time. Note, however, that it is not a polynomial time algorithm in terms of the input size, since the input N can be represented using only $O(\log N)$ bits, while the algorithm's running time depends linearly on N, which is exponential in the length of its binary representation. In fact, for arbitrary coin denominations, the change-making problem is known to be NP-hard, meaning that no polynomial-time algorithm is known (or likely exists) for the general case.

Using the table $b[1 : N]$, an optimal solution can be recovered by the following recursive procedure. It must initially be invoked with the parameters (N, b).

```
Pay(i,b)
if i>0 then
  print b[i]
  Pay(i-b[i],b)
```

Approximation algorithms

How should we design algorithms for problems where polynomial time is probably an unattainable goal? In this part, we focus on a new theme related to this question: approximation algorithms, which run in polynomial time and find solutions that are guaranteed to be close to optimal. There are two key words to notice in this definition: close and guaranteed. We will not be seeking the optimal solution, and as a result, it becomes feasible to aim for a polynomial running time. At the same time, we will be interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum.

Load balancing

We formulate the load balancing problem as follows. We are given a set of m machines $M_1, M_2, \ldots, M_m$ and a set of n jobs $a_1, a_2, \ldots, a_n$; each job a_i has a processing time t_i. We seek to assign each job to one of the machines so that the loads placed on all machines are as "balanced" as possible. More concretely, in any assignment of jobs to machines, let A_j denote the set of jobs assigned to machine M_j; under this assignment, machine M_j needs to work for a total time of

$$T_j = \sum_{i \in A_j} t_i,$$

and we declare this to be the load on machine M_j. We seek to minimize a quantity known as the makespan; it is simply the maximum load on any machine, $T = \max_j T_j$.

Although we will not prove this, the scheduling problem of finding an assignment of minimum makespan is NP-hard. We will develop an approximation algorithm for which we are always guaranteed to be within a factor of 3/2 of the optimum.

Assume that the jobs are sorted in decreasing order of their processing

time (if they are not, sort them first):

$$t_1 \geqslant t_2 \geqslant \cdots \geqslant t_n.$$

The jobs are assigned to machines in this order. For all $1 \leqslant i \leqslant n$, assign job a_i to the machine M_j whose current load (the sum of the processing time of all tasks assigned to M_j so far) is minimal.

Let T denote the maximal load in the assignment produced by this algorithm, and let T^* denote the maximal load in an optimal solution. We show that $T \leqslant \frac{3}{2}T^*$.

If $n \leqslant m$, then the statement is trivially true; in this case, our algorithm produces an optimal assignment. Thus, assume $n > m$. Let M_k be the machine whose load is maximal using the assignment produced by this algorithm, and let a_l be the job assigned last to this machine by the algorithm. If a_l is the only job assigned to M_k by the algorithm then the statement is again trivially true; in this case, our algorithm produces an optimal solution. Thus, assume that our algorithm assigned at least two jobs to M_k. Since our algorithm assigns the first m jobs to m different machines, we have $l \geqslant m+1$.

Consider the moment, when our algorithm assigns job a_l to machine M_k. At this moment M_k has a minimal load, namely $T_k - t_l$, and the load of all other machines is not smaller than this value. The loads of the machines will not decrease thereafter, so at the end of the algorithm

$$T_j \geqslant T_k - t_l$$

for all $1 \leqslant j \leqslant m$. Summing up these inequalities, we obtain

$$\sum_{j=1}^{m} T_j \geqslant m(T_k - t_l),$$

or, by rearranging,

$$T_k - t_l \leqslant \frac{1}{m}\sum_{j=1}^{m} T_j.$$

The sum of the loads of all machines must be equal to the sum of the processing times of all jobs, i.e.,

$$\sum_{j=1}^{m} T_j = \sum_{i=1}^{n} t_i.$$

Substituting this into the previous inequality, we obtain

$$T_k - t_l \leqslant \frac{1}{m}\sum_{i=1}^{n} t_i.$$

Consider the right-hand side. This is the total processing time, divided by m, i.e., the average load. No matter how we assign the jobs to the machines, there must be a machine whose load is at least this average. This implies

$$T^* \geqslant \frac{1}{m} \sum_{i=1}^{n} t_i.$$

Combining this with the previous inequality, we get

$$T_k - t_l \leqslant T^*.$$

Now, consider the first $m+1$ jobs. Since the jobs are sorted in decreasing order of their processing times, each of these jobs has processing time at least t_l. Because we have only m machines available, at least two of the first $m+1$ jobs must be assigned to the same machine in any assignment. Thus, the load of that machine is at least $2t_l$. This means that

$$T^* \geqslant 2t_l,$$

or, by rearranging,

$$t_l \leqslant \tfrac{1}{2} T^*.$$

To sum up

$$(T_k - t_l) + t_l \leqslant T^* + \tfrac{1}{2} T^*,$$

that is,

$$T_k \leqslant \tfrac{3}{2} T^*.$$

Since $T = T_k$, the claim follows.

By a more sophisticated analysis, it can be shown that

$$T \leqslant \tfrac{4}{3} T^*$$

also holds for the assignment produced by the algorithm.

Facility location

Consider the following scenario. Let P be a set of n sites, say, n small towns in upstate New York. We wish to select k centers at which to build large shopping malls. We expect that the residents in each of these n towns will shop at one of the malls, therefore we seek to choose the locations of the k malls to make them as central as possible.

More formally, we are given a positive integer k, a set P of $n > k$ sites (corresponding to the towns), and a distance function d. When we consider

instances where the sites are points in the plane, the distance function is the standard Euclidean distance between points, and any point in the plane is an option for placing a center. The algorithm we develop, however, can be applied to more general notions of distance. In fact, we can allow any distance function d satisfying the following properties:

- $d(x, y) \geqslant 0$,
- $d(x, y) = 0$ if and only if $x = y$,
- $d(x, y) = d(y, x)$,
- $d(x, y) + d(y, z) \geqslant d(x, z)$.

We assume that the residents in a given town will shop at the closest mall. Accordingly, we define the distance from a site $p \in P$ to a set of centers C as $d(p, C) = \min_{c \in C} d(p, c)$. We say that C forms an r-cover if every site is within distance at most r of some center, that is, if $d(p, C) \leqslant r$ for all sites $p \in P$. The minimum r for which C is an r-cover is called the covering radius of C, and is denoted by $r(C)$. In other words, the covering radius is the maximum distance a resident must travel to reach their nearest mall. Our goal is to select a set C of k centers that minimizes $r(C)$.

Although we will not prove this, this center-selection problem is NP-hard. We will develop an approximation algorithm for which we are always guaranteed to be within a factor of 2 of the optimum. An interesting feature of the algorithm is that the centers in C are chosen from the sites in P.

```
FacilityLocation(P,k)
Let p be any site in P
C={p}
while |C|<k do
   let p be a site in P, such that d(p,C) is maximal
   C=C∪{p}
return C
```

We prove that the set C produced by this algorithm is within a factor of 2 of the optimum. Let $P = \{p_1, p_2, \ldots, p_n\}$, let $C^* = \{c_1, c_2, \ldots, c_k\}$ be an optimal solution to the problem and let $r = r(C^*)$. Assume, for contradiction, that $r(C) > 2r$. Then there exists a site $p_{i_0} \in P$ that is farther than $2r$ from all centers in C; i.e., $d(p_{i_0}, C) > 2r$.

For each $1 \leqslant j \leqslant k$, let p_{i_j} be the site in P that is added to C by the algorithm in the jth step. Now, for each $2 \leqslant j \leqslant k$,

$$d(p_{i_j}, \{p_{i_1}, \ldots, p_{i_{j-1}}\}) \geqslant d(p_{i_0}, \{p_{i_1}, \ldots, p_{i_{j-1}}\}) \geqslant d(p_{i_0}, C) > 2r.$$

Thus, any two points of the set $\{p_{i_1}, \ldots, p_{i_k}\}$ are at distance greater than $2r$. In fact, even more is true: any two points in the set $\{p_{i_0}, p_{i_1}, \ldots, p_{i_k}\}$ are at distance greater than $2r$.

Next, consider the set C^*. Since $r(C^*) = r$, for each $0 \leqslant j \leqslant k$ there exists a center in C^* whose distance from p_{i_j} is at most r. On the other hand, no center $c_s \in C^*$ can have distance at most r to two different points in $\{p_{i_0}, p_{i_1}, \ldots, p_{i_k}\}$, say p_{i_g} and p_{i_h}, otherwise

$$2r = r + r \geqslant d(p_{i_g}, c_s) + d(p_{i_g}, c_s) \geqslant d(p_{i_g}, p_{i_h}) > 2r,$$

which is impossible.

This implies that $|C^*| \geqslant |\{p_{i_0}, p_{i_1}, \ldots, p_{i_k}\}| = k+1$, which is a contradiction.

Approximate knapsack

In the knapsack problem, we are given a set of n items $t_1, t_2, \ldots, t_n$. Each item t_i has a weight w_i and a value v_i, where w_i and v_i are positive integers. We are also given a knapsack of capacity W, which is also a positive integer. Our goal is to select a subset of the items so that the sum of their values is maximal, while the sum of their weights does not exceed the capacity W of the knapsack. Although the knapsack problem is NP-hard, we will show that for any positive real number ε, there exists a polynomial-time algorithm producing a solution whose total value is at least $1/(1+\varepsilon)$ times the value of the optimal solution.

First, we present an alternative dynamic programming algorithm for solving the knapsack problem. The subproblems are defined as follows: For each $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant v_1 + v_2 + \cdots + v_i$ find a subset of items $t_1, t_2, \ldots, t_i$ such that the total value of the items is at least j, and their total weight is as small as possible. Let $a[i, j]$ denote the total weight in the optimal solution to this subproblem. Although none of these subproblems coincides exactly with the original knapsack problem, the solution to the original problem can easily be obtained from them: it is the largest value j such that $a[n, j] \leqslant W$.

Define the base case by setting $a[i, 0] = 0$ for all $i = 0, 1, \ldots, n$. Next, let $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant v_1 + v_2 + \cdots + v_i$, and let $\mathcal{O}$ be an optimal solution to the subproblem where the target value is j and the available items are $t_1, t_2, \ldots, t_i$.

If $j > v_1 + v_2 + \cdots + v_{i-1}$, then t_i is definitely included in $\mathcal{O}$ and $\mathcal{O}' = \mathcal{O} \setminus \{t_i\}$ must be an optimal solution to the subproblem where the target value is $\max(0, j - v_i)$ and the available items are $t_1, t_2, \ldots, t_{i-1}$. Indeed, if there

were a solution to this subproblem with total weight less than that of $\mathcal{O}'$, then adding t_i would yield a solution of total weight less than that of $\mathcal{O}$ for the original subproblem, a contradiction. Hence

$$a[i,j] = w_i + a[i-1, \max(0, j - v_i)]$$

in this case.

If $j \leqslant v_1 + v_2 + \cdots + v_{i-1}$, there are two possibilities:

- If t_i belongs to $\mathcal{O}$, then $\mathcal{O}' = \mathcal{O} \setminus \{t_i\}$ is an optimal solution to the subproblem, where the target value is $\max(0, j - v_i)$ and the available items are $t_1, t_2, \ldots, t_{i-1}$. Again, if there were a solution to this subproblem with total weight less than that of $\mathcal{O}'$, then adding t_i would yield a solution of total weight less than that of $\mathcal{O}$ for the original subproblem, a contradiction. Thus, the total weight is $w_i + a[i-1, \max(0, j - v_i)]$.
- If t_i doesn't belong to $\mathcal{O}$, then $\mathcal{O}$ is already an optimal solution to the subproblem where the target value is j and the available items are $t_1, t_2, \ldots, t_{i-1}$. Thus, the total weight is $a[i-1, j]$.

Since we do not know whether t_i belongs to $\mathcal{O}$, we consider both possibilities and choose the one with smaller total weight. Hence

$$a[i,j] = \min(a[i-1,j], w_i + a[i-1, \max(0, j - v_i)]).$$

in this case.

We fill the table $a[0:n, 0:(v_1 + v_2 + \cdots + v_n)]$, which stores the values of the recursive formula, in row-major order: first the first row from left to right, then the second row, and so on. As each entry $a[i,j]$ can be computed in constant time, and since $v_1 + v_2 + \cdots + v_n \leqslant nv^*$, where $v^* = \max_i v_i$, the algorithm runs in $O(n^2 v^*)$ time.

As before, we can trace back through the table to recover an optimal set of items corresponding to the computed values $a[i,j]$.

We now turn to the promised approximation algorithm. Without loss of generality, we may assume that the weight of each item is at most W. Indeed, an item with weight greater than W cannot belong to any feasible solution, so we can simply discard it. Let ε be a positive real number. For the sake of simplicity, suppose that the reciprocal of ε is an integer.

Let $b = \frac{\varepsilon}{2n} \max_i v_i$, and consider the following modification of the original knapsack problem. For each $1 \leqslant i \leqslant n$, define $\tilde{v}_i = \lceil v_i / b \rceil b$ as the new value of item t_i. The weights of the items and the knapsack capacity remain unchanged. Note that $v_i \leqslant \tilde{v}_i \leqslant v_i + b$ for all $1 \leqslant i \leqslant n$, so the modified

values $\tilde{v}_i$ are close to the original values v_i. Also, each $\tilde{v}_i$ is an integer multiple of b.

Next, consider a scaled version of this modified problem. For each $1 \leqslant i \leqslant n$, define $\hat{v}_i = \tilde{v}_i/b$ as the new value of item t_i. The weights of the items and the knapsack capacity remain unchanged. Note that the modified problem with values $\tilde{v}_i$ and the scaled problem with values $\hat{v}_i$ have the same set of optimal solutions, their optimal values differ by exactly a factor of b, and the scaled values $\hat{v}_i$ are integers.

Now, run the above dynamic programming algorithm using the values $\hat{v}_i$ instead of v_i, and return the set of items selected by this algorithm. Let I denote the set of indices of the selected items. First, note that $\sum_{i \in I} w_i \leqslant W$, since the item weights have not changed.

To determine the running time of the algorithm, we need $\max_i \hat{v}_i$. Let $v_j = \max_i v_i$. Then $\hat{v}_j = \max_i \hat{v}_i$, and

$$\max_i \hat{v}_i = \hat{v}_j = \lceil v_j/b \rceil = 2n\varepsilon^{-1}.$$

Consequently, the running time of the algorithm is $O(n^3\varepsilon^{-1})$, which is polynomial in n for any fixed $\varepsilon > 0$.

How close is the solution obtained by this algorithm to the optimum? Consider an arbitrary subset $\{t_i \mid i \in I^*\}$ of items, such that $\sum_{i \in I^*} w_i \leqslant W$. Since the algorithm returns an optimal solution for the modified values $\tilde{v}_i$, we have

$$\sum_{i \in I} \tilde{v}_i \geqslant \sum_{i \in I^*} \tilde{v}_i.$$

Moreover, $v_i \leqslant \tilde{v}_i \leqslant v_i + b$ for any $1 \leqslant i \leqslant n$, so

$$\sum_{i \in I^*} v_i \leqslant \sum_{i \in I^*} \tilde{v}_i \leqslant \sum_{i \in I} \tilde{v}_i \leqslant \sum_{i \in I} (v_i + b) \leqslant \left(\sum_{i \in I} v_i \right) + nb.$$

Next, let $v_j = \max_i v_i$ again. Then $\tilde{v}_j = \lceil v_j/b \rceil b = 2n\varepsilon^{-1}b$. Since each weight $w_i \leqslant W$, it follows that

$$\sum_{i \in I} \tilde{v}_i \geqslant \tilde{v}_j = 2n\varepsilon^{-1}b.$$

Combining this with the previous inequality:

$$\sum_{i \in I} v_i \geqslant \left(\sum_{i \in I} \tilde{v}_i \right) - nb \geqslant 2n\varepsilon^{-1}b - nb = (2\varepsilon^{-1} - 1)nb.$$

Hence, for any $0 < \varepsilon \leqslant 1$,

$$nb \leqslant \varepsilon \sum_{i \in I} v_i,$$

and therefore

$$\sum_{i \in I^*} v_i \leqslant \left(\sum_{i \in I} v_i \right) + nb \leqslant (1 + \varepsilon) \sum_{i \in I} v_i,$$

which proves the desired approximation guarantee.

Randomized algorithms

Randomization and probabilistic analysis are themes that cut across many areas of computer science, including algorithm design. When one thinks about random processes in the context of computation, it is usually in one of two distinct ways. One view is to consider the world as behaving randomly: we study traditional algorithms that confront randomly generated input. This approach is often termed average-case analysis, since we analyze the behavior of an algorithm on an "average" input, rather than on a worst-case input. A second view is to consider algorithms that behave randomly: the world provides the same worst-case input as always, but we allow the algorithm to make random decisions as it processes the input. In this approach, the role of randomization is purely internal to the algorithm and does not require new assumptions about the nature of the input. It is the notion of a randomized algorithm that we will be considering here.

Why might it be useful to design an algorithm that is allowed to make random decisions? One answer is that by allowing randomization, we have made our underlying model more powerful. Efficient deterministic algorithms that always yield the correct answer are a special case of efficient randomized algorithms that only need to yield the correct answer with high probability (so-called Monte Carlo algorithms). They are also a special case of randomized algorithms that are always correct, and run efficiently in expectation (so-called Las Vegas algorithms). Even in a worst case world, an algorithm that performs its own "internal" randomization may be able to offset certain worst-case phenomena. Thus, problems that may not be solvable by efficient deterministic algorithms may still be amenable to efficient randomized algorithms.

Freivalds' verification of matrix multiplication

The multiplication of two matrices over a field is one of the most basic mathematical tasks. The execution of the school algorithm for matrix multiplication requires $O(n^3)$ elementary arithmetic operations. Based on the divide-

and-conquer design method, Strassen developed an $O(n^{\log_2 7})$ algorithm for matrix multiplication. Further developments have led to a sequence of improvements, and the currently best-known algorithm for matrix multiplication, due to Alman and Williams, runs in $O(n^{2.37286})$ time.

The task considered here is slightly simpler than that of general matrix multiplication. Given three $n \times n$ matrices A, B, and C, the goal is to decide whether $AB = C$. A naive deterministic approach to this equivalence problem is to compute AB, and then compare the result with C. The complexity of this approach is asymptotically the same as the complexity of multiplying A and B. Our aim is to design a randomized $O(n^2)$ Monte-Carlo algorithm for this equivalence problem. The idea is due to Freivalds.

Choose a bit vector $\alpha \in \{0,1\}^n$ uniformly at random, and compute the vectors $\beta = A(B\alpha)$ and $\gamma = C\alpha$. If $\beta = \gamma$, the algorithm returns $AB = C$; otherwise, it returns $AB \neq C$.

Clearly, the algorithm performs only $O(n^2)$ elementary arithmetic operations. Furthermore, if $AB = C$, then $\beta = \gamma$ for any $\alpha \in \{0,1\}^n$, so the algorithm always returns the correct answer in this case.

What happens when $AB \neq C$? We show that in this case, $\beta \neq \gamma$ for at least half of the bit vectors $\alpha \in \{0,1\}^n$. Thus, if α is chosen randomly, the probability of error is at most $1/2$. By repeating the process several times with independently chosen random bit vectors, the probability of error can be reduced to an arbitrarily small level.

Suppose $AB \neq C$, but for some $\alpha \in \{0,1\}^n$, we have $A(B\alpha) = C\alpha$. Then $(AB - C)\alpha$ is the zero vector. Since $AB - C$ is not the zero matrix, it has at least one non-zero entry. Let d be such an entry, say, at the intersection of the ith row and the jth column. Now consider the vector $\alpha' \in \{0,1\}^n$ obtained by flipping the jth coordinate of α: if it was 0, make it 1, if it was 1, make it 0. Then $(AB - C)\alpha'$ is not the zero vector, since its ith coordinate differs from the ith coordinate of $(AB - C)\alpha$ by d. Therefore $A(B\alpha') \neq C\alpha'$.

To complete the proof it is enough to observe that for different vectors α satisfying $A(B\alpha) = C\alpha$, we construct different vectors α' for which $A(B\alpha') \neq C\alpha'$, since we flip the same coordinate in each case.

Nuts and bolts

We are given a box containing n bolts of different sizes and another box containing n nuts of different sizes. There is a one-to-one correspondence between the nuts and the bolts, i.e., for each bolt in the first box there is a matching nut in the second box. Unfortunately, we cannot compare two bolts directly or two nuts directly. We can only determine whether the outer

diameter of a bolt is smaller than, larger than, or equal to the inner diameter of a nut (by attempting to fit the nut onto the bolt). Can one design a simple, efficient algorithm that finds the matching pairs of nuts and bolts?

While deterministic algorithms achieving $O(n \log n)$ worst-case time do exist, they are highly technical and quite involved. In contrast, the following simple Las Vegas algorithm, based on the divide-and-conquer paradigm, finds all matching pairs in $O(n \log n)$ expected time, offering a much more practical solution.

First, we randomly choose a bolt b. Next, we find the nut m that matches b, while simultaneously partitioning all other nuts into two groups: those smaller than m and those larger than m. Using the matched nut m, we then partition the bolts into two similar groups: bolts smaller than b and bolts larger than b. (The total number of comparisons in this step is $2n-1$, cf. the partition procedure in quicksort.) Finally, we recursively create the pairings by matching bolts smaller than b with nuts smaller than m, and bolts larger than b with nuts larger than m, following the same procedure.

How can we determine the expected number of comparisons in this algorithm? Without loss of generality, we may assume that the diameters of the bolts (and nuts) are the integers $1, 2, \ldots, n$. Let $T(n)$ denote the expected number of comparisons. Clearly, $T(0) = T(1) = 0$.

Let $T(n, j)$ denote the expected number of comparisons when the first chosen bolt has diameter j. Evidently,

$$T(n, j) = 2n - 1 + T(j - 1) + T(n - j).$$

Since the bolt is chosen uniformly at random,

$$T(n) = \frac{1}{n} \left(T(n, 1) + T(n, 2) + \cdots + T(n, n-1) + T(n, n) \right).$$

Using the previous equations, we can eliminate the $T(n, j)$ values:

$$T(n) = 2n - 1 + \frac{2}{n} \left(T(0) + T(1) + T(2) + \cdots + T(n-1) \right).$$

Multiplying both sides by n,

$$nT(n) = n(2n - 1) + 2(T(0) + T(1) + T(2) + \cdots + T(n-1)).$$

Writing the same formula for $n - 1$,

$$(n-1)T(n-1) = (n-1)(2n-3) + 2(T(0) + T(1) + T(2) + \cdots + T(n-2)).$$

Subtracting these two equations gives

$$nT(n) - (n-1)T(n-1) = 4n - 3 + 2T(n-1),$$

which can be rearranged as

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{4n-3}{n(n+1)}.$$

Since we are only interested in an upper bound, we can simplify by slightly overestimating the second term:

$$\frac{T(n)}{n+1} < \frac{T(n-1)}{n} + \frac{4}{n}.$$

Substituting recursively, we obtain

$$\frac{T(n)}{n+1} < \frac{4}{n} + \frac{4}{n-1} + \cdots + \frac{4}{3} + \frac{4}{2} + \frac{4}{1}.$$

Now, using

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n} < 1 + \int_1^n \frac{dx}{x} = 1 + \ln n$$

(lower Riemann sum) we conclude that

$$T(n) = O(n \log n).$$

Recommended readings

Appetizers

- Ausiello, G., Petreschi, R. (eds): *The power of algorithms.* Springer, 2013.
- Cormen, T. H.: *Algorithms unlocked.* MIT Press, 2013.
- Edmonds, J.: *How to think about algorithms.* Second edition. Cambridge University Press, 2024.
- Hromkovič, J.: *Algorithmic adventures.* Springer, 2009.
- Vöcking, B. et al (eds): *Algorithms unplugged.* Springer, 2010.

General textbooks

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to algorithms.* Fourth edition. MIT Press, 2022.
- Dasgupta, S., Papadimitriou, C., Vazirani, U.: *Algorithms.* McGraw-Hill, 2006.
- Goodrich, M. T., Tamassia, R.: *Algorithm design and applications.* Wiley, 2014.
- Horowitz, E., Sahni, S., Rajasekaran, S.: *Computer algorithms.* Second edition. Silicon Press, 2008.
- Hromkovič, J.: *Algorithmics for hard problems.* Second edition. Springer, 2004.
- Kleinberg, R., Tardos, É.: *Algorithm design.* Addison-Wesley, 2006.

- Kozen, D. C.: *The design and analysis of algorithms.* Springer, 1992.
- Levitin, A. V.: *Introduction to the design and analysis of algorithms.* Third edition. Addison-Wesley, 2011.
- Mehlhorn, K., Sanders, P.: *Algorithms and data structures: the basic toolbox.* Springer, 2008.
- Neapolitan, R. E.: *Foundations of algorithms.* Fifth edition. Jones & Bartlett Learning, 2013.
- Sedgewick, R., Flajolet, P.: *An introduction to the analysis of algorithms.* Second edition. Addison-Wesley, 2013.
- Sedgewick, R., Wayne, K.: *Algorithms.* Fourth edition. Addison-Wesley, 2011.
- Skiena, S. S.: *The algorithm design manual.* Third edition. Springer, 2020.
- Stinson, D. R.: *Techniques for designing and analyzing algorithms.* CRC Press, 2021.
- Wilf, H. S.: *Algorithms and complexity.* Second edition. A K Peters, 2002.

Stable matching

- Gusfield, D., Irving, R. W. : *The stable marriage problem: structure and algorithms.* The MIT Press, 1989.
- Knuth, D. E.: *Stable marriage and its relation to other combinatorial problems.* CRM Proceedings and Lecture Notes, American Mathematical Society, 1997.
- Manlove, D. F.: *Algorithmics of matching under preferences.* World Scientific, 2013.

Geometric algorithms

- Boissonnat, J. D., Yvinec, M.: *Geometric algorithms.* Cambridge University Press, 1998.

- de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: *Computational geometry: algorithms and applications.* Third edition. Springer, 2008.
- Devadoss, S. L., O'Rourke, J.: *Discrete and computational geometry.* Second edition. Princeton University Press, 2025.
- O'Rourke, J.: *Art gallery theorems and algorithms.* Oxford University Press, 1987.
- O'Rourke, J.: *Computational geometry in C.* Second edition. Cambridge University Press, 1998.
- Preparata, F. P., Shamos, M. I.: *Computational geometry: an introduction.* Corrected and expanded second printing. Springer, 1988.

Dynamic programming

- Lew, A., Mauch, H.: *Dynamic programming: a computational tool.* Springer, 2007.

String algorithms

- Crochemore, M., Hancart, C., Lecroq, T.: *Algorithms on strings.* Cambridge University Press, 2007.
- Gusfield, D.: *Algorithms on strings, trees and sequences.* Cambridge University Press, 1997.
- Kingsford, C.: *An introduction to string algorithms.* Princeton University Press, 2026.

Knapsack problem

- Kellerer, H., Pferschy, U., Pisinger, D.: *Knapsack problems.* Springer, 2004.
- Martello, S., Toth, P.: *Knapsack problems: algorithms and computer implementations.* John Wiley and Sons, 1990.

Scheduling algorithms

- Baker, K. R., Trietsch, D.: *Principles of sequencing and scheduling.* Wiley, 2009.
- Pinedo, M. L.: *Scheduling: theory, algorithms, and systems.* Sixth edition. Springer, 2022.

Approximation algorithms

- Vazirani, U.: *Approximation algorithms.* Springer, 2001.
- Williamson, D. P., Shmoys, D. B.: *The design of approximation algorithms.* Cambridge University Press, 2011.

Randomized algorithms

- Hromkovič, J.: *Design and analysis of randomized algorithms: introduction to design paradigms.* Springer, 2005.
- Mitzenmacher, M., Upfal, E.: *Probability and computing.* Second edition. Cambridge University Press, 2017.
- Motwani, R., Raghavan, P.: *Randomized algorithms.* Cambridge University Press, 1995.

Problem solving

- Backhouse, R.: *Algorithmic problem solving.* Wiley, 2011.
- Benoit, A., Robert, Y., Vivien, F.: *A guide to algorithm design: paradigms, methods, and complexity analysis.* CRC Press, 2014.
- Bosc, P., Guyomard, M., Miclet, M.: *Algorithm design: a methodological approach — 150 problems and detailed solutions.* CRC Press, 2023.
- Izadkhah, H. *Problems on algorithms: a comprehensive exercise book for students in software engineering.* Springer, 2022.
- Laaksonen, A. *Guide to competitive programming.* Third edition. Springer, 2024.

- Levitin, A., Levitin, M.: *Algorithmic puzzles.* Oxford University Press, 2011.
- Parberry, I.: *Problems on algorithms.* Prentice Hall, 1995.
- Shen, A.: *Algorithms and programming: problems and solutions.* Second edition. Springer, 2010.
- Skiena, S. S., Revilla, M. A.: *Programming challenges: the programming contest training manual.* Springer, 2003.