CS 341

by Steven Cao

Contents

§1. Review

§1.1. Convex hull problem

Given a set of points, the goal is to find the smallest convex set that contains all the points.

Notice that each edge in the convex hull has the property that all points lie on only one side of the edge.

Algorithm 1:

For all pairs of points r, s, find the line through the points and check if all points lie on one side. If so, include it in the convex hull. This can be done by checking the sign of the cross product of the vectors rp and sp for a given point p.

There are $\Theta(n^2)$ pairs of points and for each edge, $O(n)$ points to check; each check taking $O(1)$ operations. The overall runtime is $O(n^3)$.

Algorithm 2 (Jarvis March):

Once we have a line rs, there is a natural next line st sharing a common point. To find st, choose the line through s with the minimum angle from rs. The runtime is $O(n^2)$.

In practice the number of convex edges can be much smaller than the number of points. Let h be the number of edges on the convex hull. The runtime can also be written as $O(hn)$. If h is constant this algorithm has a good runtime: $O(n)$.

Algorithm 3 (Reduction):
- Sort the points by their x-coordinate (from leftmost to rightmost).
- Traverse the points in order to build edges of the upper and lower hull. If the point is on the right side of the previous edge, add the point to the hull. Otherwise, replace the previous edge so that it points to the current point.

Sorting takes $O(n \log n)$ and building the hull takes $O(n)$. Hence, the overall runtime is $O(n \log n)$ (with the sorting step as the bottleneck). Note that if one can solve sorting faster, the convex hull problem speeds up as well.

Algorithm 4 (Divide and Conquer):
- Find the median of the points and divide them into two sides using quickselect ($O(n)$ expected).,
- Solve the problem on both sides, creating two separate convex hulls.,
- Find the edge e that contains the rightmost point of the left hull and the leftmost point of the right hull.,
- Walk along e to the left and right until the top and bottom edges are found, creating a new convex hull ($O(n)$).

The runtime can be written as

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

so $T(n) \in O(n \log n)$.

We can reduce sorting to the convex hull problem. Given n values $x_1, ..., x_n$, map each x_i to (x_i, x_i^2). The points then form a parabola, which is convex. Solving the convex hull problem returns the points in order. Since the reduction takes $O(n)$ and comparison-based sorting has a lower bound of $O(n \log n)$, this proves that the lower bound for the convex hull problem is $O(n \log n)$.

§1.2. Random access machine model

In the random access machine model, each memory location holds one word. A number a requires approximately $\log a$ bits to represent in binary. With typical methods:

- $a + b \in O(\max(\log a, \log b))$
- $a \times b \in O(\log(a + b))$

A more efficient multiplication algorithm exists with running time $O\left((\log(a + b))^{\{0.59\}}\right)$.

In C/C++ numbers generally have fixed word sizes, so we do not need to worry about the size of the numbers.

Below is an example algorithm.

In the worst case (assuming each number fits in one word), if each number is close to the maximum representable value, then Array-Product$(A, n) \in O(n^2)$.

§1.3. Asymptotic analysis

Some examples:

- $p_{d(n)} \in O\left(n^d\right)$, where $p_{d(n)}$ is a polynomial of degree d.
- $(n + 1)! \in O((n + 1)!)$,
- $\log(n!) \in \Theta(n \log n)$

Remember that Big-O provides an upper bound. (For example, an algorithm in $O(n)$ is also in $O(n^2)$.) In many cases when Big-O is specified, it is actually meant to be Θ.

§1.4. Reductions

Problem: Given an array and an integer m, find a pair of values in the array which sum to m.

§2. Graph Algorithms

Definition 2.1 (Graph).
A graph $G = (V, E)$ consists of:

- A vertex set V, with $|V| = n$.
- An edge set $E \subseteq V \times V$ with $|E| = m$, $m \leq n^2$.

Edges can be **directed** or **undirected**.

§2.1. Storing graphs

Graphs can be stored using either an adjacency matrix or an adjacency list.

Definition 2.1.1 (Adjacency Matrix).
An $n \times n$ matrix A where

$$A[i, j] = \begin{cases} 1 \ (i, j) \in E \\ 0 \text{ otherwise} \end{cases}$$

This method requires $O(n^2)$ space.

Definition 2.1.2 (Adjacency List).
An array/list of length n, where each entry corresponds to a vertex and contains a list of its neighboring vertices.

This method uses $O(n + m)$ space. (For very dense graphs it might use more space than an adjacency matrix.)

§3. Greedy algorithms

§3.1. Minimizing lateness

Suppose we are given a number of tasks to complete with times to complete t and deadlines d. We want to determine:
- Can all the tasks be completed before their deadlines?
- How to minimize the maximum lateness?

An observation: once we start job i, we might as well complete it. For example, if we do jobs ABA in this order, we can complete B at an earlier by doing it before A, but we will still finish A at the same time.

Since the jobs are done contiguously, and taking a break is never useful, we simply need to find an ordering of the jobs.

If we do the shortest job first, the longest jobs may get held up by the shorter jobs, pushing their completion times back where it may be more optimal to do them first. For example, if job A is long with a short deadline and job B is short with a long deadline, it is more optimal to do A.

If we do the jobs with the least amount of slack $(d_i - t_i)$, a similar problem arises.

The optimal solution is to do the jobs in order of earliest deadline first. We can do an exchange proof to show that this is the optimal solution.

Let G be the jobs in order of earliest deadline. Let O be an optimal solution.

Let $j_a, j_b \in G$ such that j_b is before j_a in O. Then, j_a has an earlier deadline than j_b. So,

$$l_G(b) = \max(d_b - t_b, 0) \le \max(d_b - t_a, 0) = l_O(b)$$

since $t_b \ge t_a$ for times t corresponding to G, and

$$l_G(a) = \max(d_a - t_a, 0) \le \max(d_b - t_a, 0) = l_O(b)$$

since $d_a \le d_b$.

So if we swap the jobs in O, the lateness will not increase. We can do this continuously until the two solutions match. Therefore, the earliest deadline first solution is optimal.

Exchange proofs like this can be used to show that greedy algorithms are optimal.

§3.2. Knapsack problems

Given a set of n items with weights w and values v, and a knapsack with a maximum weight W. We want to determine the maximum value of items that can be placed in the knapsack.

A feasible solution is a tuple $X = [x_1, ..., x_n]$ where

$$\sum_{i=1}^{n} w_i x_i \leq W$$

In the 0-1 knapsack problem, we require each item to be either taken or not ($x_i \in \{0, 1\}$). In the fractional knapsack problem, we can take fractions of items ($x_i \in [0, 1] \subseteq \mathbb{Q}$).

Some possibly greedy strategies:
- Take items in decreasing order of value
- Take items in increasing order of weight
- Take items in decreasing order of value-to-weight ratio $\frac{v_i}{w_i}$

The value-to-weight ratio strategy produces the optimal solution.

Assume items are ordered by $\frac{v_i}{w_i}$. Let the greedy solution be $[x_1, x_2, ..., x_k, ..., x_l, ..., x_n]$. Let an optimal solution be $[y_1, y_2, ..., y_k, ..., y_l, ..., y_n]$, which mathces the greedy solution on M indices. Here we define k to be the first index where the two solutions differ.

If $M = n$ then the two solutions are the same, so the greedy solution is optimal.

Otherwise, we know $x_k > y_k$ because the greedy strategy would take as much of the kth item as possible. Since the total weight of both solutions is the same (W), there must be a later item l such that $x_l < y_l$.

We can shift the weight between the kth and lth items in the optimal solution. The new values are

$$y'_k = y_k + \Delta$$
$$y'_l = y_k - \Delta$$

where

$$\Delta = \min(x_k - y_k, y_l - x_l)$$

Then, $y'_k = x_k$ or $y'_l = x_l$. The new optimal solution matches the greedy solution on at least $M + 1$ indices.

§3.3. Stable marriage

Given a set of n co-op students $S = [s_1, ..., s_n]$ and a set of n employers $E = [e_1, ..., e_n]$. Each student ranks the employers in order of preference, and each employer ranks the students in order of preference. We want to find a stable matching, where no student and employer prefer each other over their current matches.

The Gale-Shapley algorithm is a greedy algorithm that finds a stable matching.

§4. Dynamic programming

The main idea of dynamic programming is to solve the subproblems for smaller to larger (bottom up) and store the results as it is done.

Consider the recursive Fibonacci algorithm:

```
algorithm Fib($n$)
    if $n = 0$ or $n = 1$
        return $n$
    else
        return Fib($n - 1$) + Fib($n - 2$)
```

This computes the same subproblems multiple times, which is inefficient. Instead we can save the results of the subproblems and use them when needed.

```
algorithm Fib($n$)
    $f[0] \leftarrow 0$
    $f[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$
        $f[i] \leftarrow f[i-1] + f[i-2]$
    return $f[n]$
```

§4.1. Text segmentation

Given a string A of length n. We want to determine whether A can be split into 2 or more words.

The brute force solution is to try all possible splits and check if the substrings are words. This takes $O(2^n)$ time.

We can use greedy straregy like maximal munch to find solutions, but this won't work for all cases.

We can build up a solution for $A[n]$ from smaller subproblems. Suppose we know Split(k) for all $k < n$ where

$$\text{Split}(k) = \begin{cases} \text{true} & A[1, ..., k] \text{ is splittable} \\ \text{false otherwise} \end{cases}$$

To find Split(n), try Split(j) and Word($j + 1, n$) for all $j < n$.

```
algorithm Splittable($A$, $n$)
    Split[0] $\leftarrow$ true
    for $k \leftarrow 1$ to $n$
        Split[$k$] $\leftarrow$ false
        for $i \leftarrow 0$ to $k - 1$
            if Split[$i$] and Word($j + 1, k$)
                Split[$i$] $\leftarrow$ true
    return Split[$n$]
```

The runtime is $O(n^2)$.

§4.2. Longest increasing subsequence

Given a sequence of numbers A. We want to find the longest subsequence of A that is increasing.

We can solve with a similar approach to the previous problem.

Define LISe$[k]$ to be the length of the longest increasing subsequence of $A[1..k]$ that ends with $A[k]$.

To compute LISe$[k]$, consider all previous longest increasing subsequences that can be extended by $A[k]$.

```
algorithm LIS($A$, $n$)
    LISe[1] $\leftarrow 1$
    for $k \leftarrow 2$ to $n$
        LISe[$k$] $\leftarrow 1$
        for $j \leftarrow 1$ to $k - 1$
            if $A[k] > A[j]$
                LISe[$k$] $\leftarrow \max(\text{LISe}[k], 1 + \text{LISe}[j])$
    return $\max(\text{LISe}[1], ..., \text{LISe}[n])$
```

The runtime is $O(n^2)$.

If we want to find the actual subsequence, we can store the previous index that was used to extend the subsequence. Then, from the maximum length, we can backtrack through the previous index array to find the subsequence.

§4.3. Longest common subsequence

Given two strings x, y. We want to find the longest subsequence of characters in x and y which are in order (but doesn't need to be contiguous).

Let $M[i, j]$ be the length of the longest common subsequence of $x[1..i]$ and $y[1..j]$.

We want to build up the table until we reach $M[n, m]$.

Given previous values of M, we can compute $M[i, j]$ by

$$M[i, j] = \begin{cases} 1 + M[i - 1, j - 1] & \text{if } x[i] = y[j] \\ \max(M[i - 1, j], M[i, j - 1]) & \text{otherwise} \end{cases}$$

We can build up the table row by row.

```
algorithm LCS-Length($x, y$)
    $n \leftarrow |x|$
    $m \leftarrow |y|$
    $M[i, 0] \leftarrow 0 \quad \forall i \in [1, n]$
    $M[0, j] \leftarrow 0 \quad \forall j \in [1, m]$
    for $i \leftarrow 1$ to $n$
        for $j \leftarrow 1$ to $m$
            if $x[i] = y[j]$
                $M[i, j] \leftarrow 1 + M[i - 1, j - 1]$
            else
                $M[i, j] \leftarrow \max(M[i - 1, j], M[i, j - 1])$
    return $M[n, m]$
```

We need every subproblem in order to compute $M[m, n]$. The runtime is $O(nm)$.

If we want to find the actual subsequence, we can backtrack through the table separetely.

```
algorithm LCS-Sequence($i, j, M$)
    if $i = 0$ or $j = 0$
        done
    else if $x[i] = y[j]$
        return LCS-Sequence($i - 1, j, M$) $\|$ $x[i]$
    else if $M[i, j] = M[i - 1, j]$
        return LCS-Sequence($i - 1, j, M$)
    else if $M[i, j] = M[i, j - 1]$
        return LCS-Sequence($i, j - 1, M$)
```

This takes $O(n + m)$ time.

Longest common subsequence is used in diff tool, which is commonly used in version control systems for example.

§4.4. Edit distance

Given two strings x, y, we can convert x to y by making one-character changes:

- Add a letter
- Delete a letter
- Replace a letter

Again we can use a table $M[i, j]$ to represent the edit distance between $x[1..i]$ and $y[1..j]$. Given costs a, d, r for each operation, we can compute $M[i, j]$ by

$$M[i,j] = \begin{cases} jd & \text{if } i = 0 \\ ia & \text{if } j = 0 \\ M[i-1, j-1] & \text{if } x[i] = y[j] \\ \min \begin{cases} r + M[i-1, j-1] & \text{replace } x[i] \text{ with } y[j] \\ d + M[i-1, j] & \text{delete } x[i] \\ a + M[i, j-1] & \text{add } y[j] \end{cases} & \text{otherwise} \end{cases}$$

§4.5. Weighted interval scheduling

Problem 4.5.1 (Weighted interval scheduling).
Given a set of intervals I with starting and ending times and weights $w : I \to \mathbb{R}$, we want to find the set $S \subseteq I$ which maximizes total weight.

A more general version of the problem is the maximum weight independent set problem.

Problem 4.5.2 (Maximum weight independent set).
Given set of items I, weights w and set C of conflicting pairs of items. We want to find the set $S \subseteq I$ which maximizes total weight and has no conflicting pairs of items.

This can be modelled as a graph problem where each item is a vertex and conflicting pairs are edges. This is an NP-complete problem, but interval scheduling can be solved in polynomial time.

For weighted interval scheduling, let $M[i]$ represent the max weight subset of intervals $1..i$. We can either choose to include interval i or not.

$$M[i] = \max\begin{cases} M[i-1] & \text{don't choose } i \\ w(i) + M[X] & \text{choose } i \end{cases}$$

§4.6. Optimal binary search tree

Problem 4.6.1 (Optimal binary search tree).
Given a set of items $I = \{1, ..., n\}$ and probabilities $p_1, ..., p_n$ that item i will be searched. Find a BST which minimizes the expected search time

$$\sum_{i \in I} p_i(1 + \text{Depth}(i))$$

We can try all choices for the root node. Suppose the root node is k. Assuming the items are sorted, the left subtree should be optimal for items $1, ..., k-1$ and the right subtree should be optimal for items $k+1, ..., n$.

Let $M[i, j]$ be the optimal BST for items $i, ..., j$. Then,

$$M[i, j] = \min_{k=i..j} \{M[i, k-1] + M[k+1, j]\} + \sum_{t=i}^{j} p_t$$

§4.7. 0-1 knapsack

Problem 4.7.1 (0-1 knapsack).
Given a set of items $\{1, ..., n\}$ where each item i has weight w_i and values v_i, and a knapsack with capacity W. Find a subset of items S such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.

Unlike fractional knapsack, items are not divisible.

For subproblems, consider
- Limit items available to $1, ..., i$
- Keep track of remaining capacity

So, we can define subproblems as $M[i, w]$ representing the maximum value of items $1, ..., i$ with remaining capacity $w \leq W$.

If we want to take item i, we need the solution to subproblem $M[i-1, w-w_i]$. Or if we don't, we need $M[i-1, w]$

The bases cases are
- $M[0, w] = 0$
- $M[i, 0] = 0$

We can iterate over i and w to compute the table.

```
algorithm 0-1-Knapsack($n, W$)
    for $i \leftarrow 1$ to $n$
        $M[i, 0] \leftarrow 0$
        for $w \leftarrow 1$ to $W$
            if $w_i > w$
                $M[i, w] \leftarrow M[i-1, w]$
            else
                $M[i, w] \leftarrow \max(M[i-1, w], v_i + M[i-1, w-w_i])$
    return $M[n, W]$
```

This takes $O(nW)$ time. This is pseudo-polynomial time: the runtime depends on the numeric value of an input W.

To recover the solution from the table, we can backtrack and choose the path depending on whether the item was taken or not ($M[i, w] = M[i-1, w]$ for not taken).

```
$i \leftarrow n$
$w \leftarrow W$
$S = \emptyset$
while $i > 0$ and $w > 0$
    if $M[i, w] = M[i-1, w]$
        $i \leftarrow i - 1$
    else
        $S \leftarrow S \cup \{i\}$
        $i \leftarrow i - 1$
        $w \leftarrow w - w_i$
return $S$
```

Suppose W requires k bits to store where $k \in \Theta(\log W)$. Then, the runtime can be written as $O(n2^k)$. Usually this is not a problem for n because we expect each item to fit in a word, but W can be very large since it can be proportional to the sum of all weights.

If we solve the problem top-down, we don't actually need to store the entire table.

§4.8. Memoization

When we solve a problem top-down recursively, we can store the results of subproblems in a table. If the same subproblem is encountered again, we can look up the result in the table instead of recomputing it.

The advantage is that, regardless of problem and the order of subproblems in our solution, memoization can improve the runtime of the algorithm. The disadvantage is that it is harder to analyze runtime, and recursion adds extra overhead.

§4.9. Shortest paths in DAG

Topological sort is very useful for solving shortest path problems in DAGs and many other DAG problems.

Dijkstra's algorithm can be used to solve SSSP in a graph with non-negative weights. The Bellman-Ford algorithm is a DP algorithm that can solve SSSP in a graph with negative weights.

Let $d_i(v)$ be the weight of the shortest path from s to v using $\leq i$ edges. Then,

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(sv) & \text{if } sv \in E \\ \infty & \text{otherwise} \end{cases}$$

we want to compute $d_{n-1}(v)$.

$$d_i(v) = \min \begin{cases} d_{i(v)} & \leq i-1 \text{ edges} \\ \min_u \{d_{i-1}(u) + w(uv)\} & i \text{ edges} \\ \infty & \text{otherwise} \end{cases}$$

For the actual algorithm, we can reuse the same table for all d_i.

```
algorithm Bellman-Ford($G, s$):
    for $v \in V$
        $d[v] \leftarrow \infty$
    $d[s] \leftarrow 0$
    for $i \leftarrow 1$ to $n - 1$
        for $(u, v) \in E$
            $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$
    return $d$
```

The algorithm takes $O(n(n+m))$ time.

For all pair shortest paths (Floyd-Warshall), let $D_i[u, v]$ represent the length of the shortest uv -path using intermediate vertices in $1, ..., i$. Then,

$$D_i[u, v] = \min \begin{cases} D_{i-1}[u, i] + D_{i-1}[i, v] & \text{use vertex } i \\ D_{i-1}[u, v] & \text{don't use } i \end{cases}$$

with base cases

$$D_0[u, v] = \begin{cases} 0 & \text{if } u = v \\ w(u, v) & \text{if } uv \in E \\ \infty & \text{otherwise} \end{cases}$$

We can again reuse the same table for all D_i.

```
algorithm Floyd-Warshall($G$):
    for $u \in V$
        for $v \in V$
        if u = v
            $D[u, v] \leftarrow 0$
        else if u v in E
            $D[u, v] \leftarrow w(u, v)$
        else
            $D[u, v] \leftarrow \infty$
    for $i \in [1, n]$
        for $u \in V$
            for $v \in V$
                $D[u, v] \leftarrow \min(D[u, v], D[u, i] + D[i, v])$
    return $D[n]$
```

§5. Intractable problems

§5.1. Backtracking

Backtracking is a systematic way to try all possible solutions, similar to searching an implicit tree of partial solutions. It is used in decision problems.

Problem 5.1.1 (Subset sum).
Given a set of weights S and a target weight W, determine if there is a subset of S that sums to W.

This problem is NP-complete.

To solve these problems, we can use a recursive backtracking algorithm similar to DFS.

§5.2. Branch and bound

If we can determine the lower bound of a solution, we can use branch and bound to prune the search space.

§5.3. Decision problems

Definition 5.3.1 (Decision problem).
Given a problem instance I, answer a certain question with either "yes" or "no".

Definition 5.3.2 (Class P).
An problem which has a worst-case runtime of $O(n^k)$ for some constant k.

Class P is the set of problems solvable in polynomial time. There are many problems for which there are no known polynomial time algorithms to solve the problem, or also check if a solution is correct.

Definition 5.3.3 (Class NP).
There exists a certificate C such that a solution can be verified in polynomial time.

Theorem 5.3.1.

$$\mathrm{P} \subseteq \mathrm{NP}$$

Definition 5.3.4 (coNP).
The class of decision problems where the answer "no" can be verified in polynomial time.

Definition 5.3.5 (NP-Complete).
A decision problem X is NP-Complete if $X \in \text{NP}$ and for every $Y \in \text{NP}$, $Y \geq_P X$.

Implication of X being NP-Complete:
- If X can be solved in polynomial time, then all NP problems can be solved in polynomial time. ($X \in \text{P} \implies \text{P} = \text{NP}$)
- If X cannot be solved in polynomial time, then no NP-complete problem can be solved in polynomial time.
- If $x \in \text{coNP}$, then $\text{NP} = \text{coNP}$.