Data Structures and Algorithms: The formal notes

Josh Felmeden

November 10, 2019

# Contents

1 Greedy algorithms

1.1 Interval scheduling

Let's suppose that you're running a satellite imaging service. Taking a satellite picture of an area isn't instant and can take some time. It can also only be done on the day where the satellite's orbit is lined up correctly. Say you want to request some images to be taken from said satellite, each of which can only be taken at certain times and you can only take one picture at a time. How do we satisfy as many requests as possible?

The requested satellite times that we have to deal with are: 12:00-12:30, 12:05-12:20, 12:15-12:55, 12:20-12:25, 12:38-12:50, and 12:45-13:00.

If we visualise this in a graph, we get:
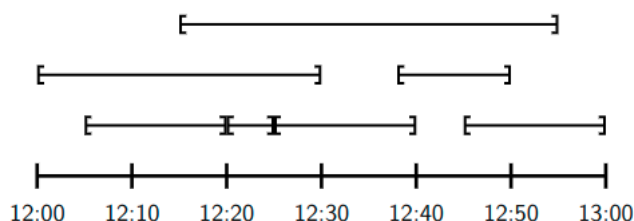
If we take a greedy algorithm approach to assigning these slots, we could do something like assign the slot that finishes earliest, and then repeat doing this until we have reached the end. For example, the slot that finishes fastest is 12:05-12:20, so we assign this. This now removes the ability for both 12:10-12:30 and 12:15-12:55 to be assigned, so we remove these. This continues until we end up with something looking like this:
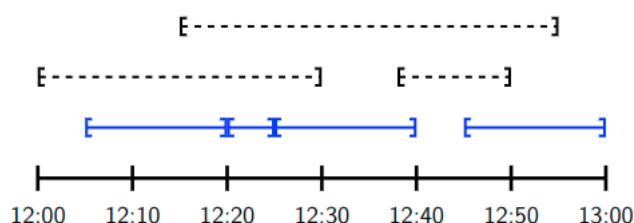
This means that we satisfy four requests (which is actually the maximum possible, so well done us).

We can formalise this by saying that a **request** is a pair of integers (s, f) with $0 \leq s \leq f$.

The algorithm that we're left with is this:

```
public sub greedySchedule
    sort R
    for each i in {1 ... n} do
        if s_i >= lastf then
            A.append(s_i, f_i)
            lastf = f_i
        end if
    next
end sub
```

Now, we need to prove that the output is actually a **compatible subset** of R. This is sort of intuitive because the set we added doesn't break compatibility, since $s \geq$ `lastf` and `lastf` is the latest finish time that's already in A.

We can formalise this with a loop invariant. At the start of the ith iteration, we see that

- A contains a compatible subset $\{(S_1, F_1), \ldots, (S_t, F_t)\}$ of R.

- `lastf` $= \max(\{0\} \cup \{F_j : j \leq t\})$

The base case ($i = 0$) is immediate because $A = []$. The induction step is that A was compatible at the start of the iteration, and therefore if we append a pair s_i, f_i to A then $s \geq$ `lastf` $\geq F_j$ for all $f \leq t$. This means that (s_i, f_i) is compatible with A.

> **Greedy algorithms definition**
>
> Greedy algorithms are actually an informal term and people have different definitions. The definition that we're going to use is:
>
> - They start with a sub-optimal solution.
>
> - They look over all the possible improvements and pick the one that looks the best at the time.
>
> - They never backtrack in 'quality'.

Greedy algorithms might fail; it's not enough to just do the obvious thing at each stage. While the algorithms might fail initially, we can use the knowledge that we gained from the results of the algorithm to design a more correct one.

2 Graphs

> **Graph Definition**
>
> A **graph** is a pair $G = (V, E)$ where $V = V(G)$ is a set of **vertices** $E = E(G)$ is a set of **edges** contained in $\{\{u, v\} : u, v \in V, u \neq\}$

> **Walk Definition** 5
>
> A **walk** in a graph $G = (V, E)$ is a sequence of vertices such that $\{v_i, v_{i+1} \in E$ for all $i \leq k - 1$
> We say that the walk is from v_0 to v_k and call k the length of the walk.

2.1 Euler walks

An **Euler walk** is one that contains every edge in G exactly once.

Two graphs might be **equal**. This is the case when two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are equal and written $G_1 = G_2$ if $V_1 = V_2$ and $E_1 = E_2$. This does present some issues, however, because sometimes graphs look like they should be equal, when they're not because the edges are labelled differently.

This is where **isomorphism** comes in. G_1 and G_2 are **isomorphic** if there's a bijection $f : V_1 \to V_2$ such that $\{f(u), f(v)\} \in E_2$ if and only if $\{u, v\} \in E_1$.

Intuitively, this means that $G_1 \xrightarrow{\sim} G_2$ if they are the same graph but the vertices are relabelled.

In a graph $G = (V, E)$, the **neighbourhood** of a vertex v is the set of vertices joined to v by an edge. Formally, $N_G(v) = \{w \in V : \{v, w\} \in E$. Also, for all sets of vertices $X \subseteq V = \cup_{v \in X} N_G(v)$

The **degree** of a vertex v is the **number** of vertices joined to v. Formally: $d_G(v) = |N_G(v)|$

Theorem: If G has an Euler walk, then either:

- Every vertex of G has even degree or

- All but two vertices v_0 and v_k have even degree, and any Euler walk must have v_0 and v_k as endpoints.

Does every single graph that satisfies both of these conditions have an Euler walk? No, because the graphs need to be **connected**.

Within a graph, we also have subgraphs and induced subgraphs. A **subgraph** $H = (V_H, E_H)$ of G is a graph with $V_H \subseteq$ and $E_H \subseteq E$. H is an **induced subgraph** if $V_H \subseteq V$ and $E_H = \{e \in E : e \subseteq V_H\}$.

For all vertex sets $X \subseteq V$, the graph is **induced** by X is:

$$G[X] = (X, \{e \in E : e \subseteq X\})$$

A **component** H of G is the maximal connected induced subgraph of G, so $H = G[V_H]$ is connected, but $G[V_H \cup \{v\}]$ is disconnected for all $v \in V \backslash V_H$.

> Theorem: Let $G = (V, E)$ be a **connected** graph, and let $u, v \in V$. Then, G has an Euler walk from u to v if and only if either:
>
> 1. $u = v$ and every vertex of G has even degree, or
>
> 2. $u \neq v$ and every vertex of G has even degree except u and v.

3 Sequential Processes

4 Fast Fourier Transform

4.1 Polynomials

A **degree** $n - 1$ polynomial in x can be seen as a function:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i$$

Any integer that's bigger than the degree of a polynomial is a *degree bound* of said polynomial. The polynomial A is:

$$a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 \cdots + a_{n-1} x^{n-1}$$

The values a_i are the *coefficients*, the degree is $n - 1$ and n is a degree bound. We're able to express any integer as some kind of polynomial by setting x to some base, say for decimal numbers:

$$A = \sum_{i=0}^{n-1} a_i \cdot 10^i$$

The variable x just allows us to evaluate the polynomial at a point. A really fast way to evaluate the polynomial is to use **Horner's Rule**.

> **Horner's Rule**
>
> Instead of computing all the terms individually, we do:
>
> $$A(3) = a_0 + 3 \cdot (a_1 + 3 \cdot (a_2 + \cdots + 3 \cdot (a_{n-1})))$$
>
> This method requires $O(n)$ operations. For example, if we consider $A(x) = 2 + 3x + 1.x^2$, we can evaluate this as:
>
> $$A(x) = 2 + x(3 + 1.x)$$

Once we have our polynomial representations, we might be doing some arithmetic with them. We're allowed to write polynomials in a *coefficient representation*. Here, the addition of $C = A + B$ constructs C as the vector:

$$(a_0 + b_0, a_1 + b_1, a_2 + b_2, \ldots, a_{n-1} + b_{n-1})$$

A and B should really have the same length, but we're allowed to just pad the coefficients with zero to make this the case.

4.1.1 Point value Representation

We know that if we're given a polynomial, we can graph it. We can use this fact to represent a polynomial as a list of its points. For point value representation, the addition $C = A + B$ constructs C as:

$$\{x_0, y_0 + z_0), (x_1, y_1 + z_1), (x_2, y_2 + z_2), \ldots, (x_{n-1}, y_{n-1} + z_{n-1})$$

where x_i is a point, $y_i = A(x_i) and z_i = B(x_i)$.

It's important to note that the two value representations **must** use the same evaluation points. Both of the operations are $O(n)$ in terms of how long they take.

4.1.2 Polynomial multiplication

Computing a polynomial multiplication (also called **convolution**) is a little harder than addition. It does, however, become much easier when we use point value representation:

$$\{x_0, y_0 \cdot z_0), (x_1, y_1 \cdot z_1), (x_2, y_2 \cdot z_2), \ldots, (x_{n-1}, y_{n-1} \cdot z_{n-1})$$

where x_i is a point, $y_i = A(x_i) and z_i = B(x_i)$.

The normal method of calculating multiplication is $O(n^2)$, while using point value representation only takes $O(n)$. So, is there an easy way to convert to point value representation? Actually, yes. What we need to do is to evaluate the polynomial to a point-value representation, multiply and finally interpolate (the opposite of evaluating) back again.

Long story short, we need to develop two fast algorithms that construct the coefficients for the point value representation and then interpolate. So the main steps to multiply the two polynomials A and B (of degree n) are:

1. *Double degree bound*: Create coefficient representations of $A(x)$ and $B(x)$ as degree bound $2n$ polynomials by adding n high-order zero coefficients to each.

2. *Evaluate*: Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of FFT of order $2n$

3. *Pointwise multiply*: compute a point-value representation of $cC(x) = A(x)B(x)$ by multiplying the values Pointwise

4. *Interpolate*: Create a coefficient representation of $C(x)$ through a single application of the *inverse* FFT.

The first and third steps are really easy to perform in $O(n)$ time. The claim is that if we evaluate at the complex roots of unity then we can perform steps 2 and 4 in $O(n \log n)$ time.

4.2 Evaluation at roots of unity

First, we need to look at evaluation. We need to evaluate a polynomial of degree n at n different points. It appears that the complexity of our method is just going to be $O(n^2)$, but there is a faster way of doing it than just using Horner's rule. This is when we evaluate the points at *special* points (namely the **N-th complex roots of unity**).

> **N-th complex roots of unity**
>
> - The roots of unity are the values $\omega_N = e^{2\pi ij/N}$ for $j = 0, 1, \ldots, N-1$
>
> - Say we are evaluating at N pints, we take the N-th complex roots of unity ω_N
>
> - This means we evaluate the polynomial at the points:
>
> $$\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega_N^{n-1}$$

We want to evaluate a polynomial A at the n roots of unity. Therefore, we evaluate:

$$A(x) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

for every $k = 0, 1, \ldots, n-1$

We define the vector of results of these evaluations as:

$$y_k = A(\omega_n^k)$$

This vector $y = (y_0, \ldots, y_{n-1})$ is the **discrete Fourier Transform** (DFT) of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$

> **The Cancellation Lemma**
>
> $$\omega_{dN}^{dk} = \omega_N^k$$

> **The Halving Lemma**
>
> If $N > 0$ is even then the squares of the N complex N-th roots of unity are the $N/2$ complex $N/2$-th roots of unity
> The proof of this is that we have $(\omega_n^k)^2 = \omega_{n/2}^k$ for any nonnegative integer k because of the cancellation lemma.

4.3 Fast Fourier Transform

The really basic idea of the Fast Fourier transform is that we define two new polynomials;

$$A^{[0]}(x) = a_0 + a_2 x + \cdots + a_{N-2} x^{N/2-1}$$
$$A^{[1]}(x) = a_1 + a_3 x + \cdots + a_{N-1} x^{N/2-1}$$

5 Dynamic Programming

We use *dynamic programming* for finding efficient algorithms for problems that can be broken down into simpler, overlapping subproblems. Basically,

1. Find a recursive formula for the problem (in terms of answers to the subproblems)

2. Write down a naive recursive algorithm

3. Speed it up by storing the solutions to the subproblems (**memoization**)

4. Derive an iterative algorithm by solving the subproblems in good order.

5.1 Weighted interval scheduling

We've seen the scheduling problem before on the course, but this is different (and as it turns out, harder).

A **schedule** is a set of compatible intervals. The *weight* of a schedule is the sum of the weight of the intervals it contains.

We could use a greedy algorithm, but it is really slow and not what we're looking for. It's not even going to cut it full stop.

How is the input provided?

The intervals are given in an array A of length n.

$A[i]$ stores a triple (s_i, f_i, w_i) which defines the ith interval.

The intervals are sorted by finish time i.e. $f_i \leq f_{i+1}$.=, or the interval i finishes before the interval $i + 1$ finishes.

Compatible Intervals

For all i, we let $p(i)$ to be the rightmost interval (in order of finish time) which finishes before the ith interval but doesn't overlap it.

In the below example, if we take $i = 7$, $p(7) = 3$ because it's the first interval that doesn't overlap with it (and 3 is the position of the interval in the array).
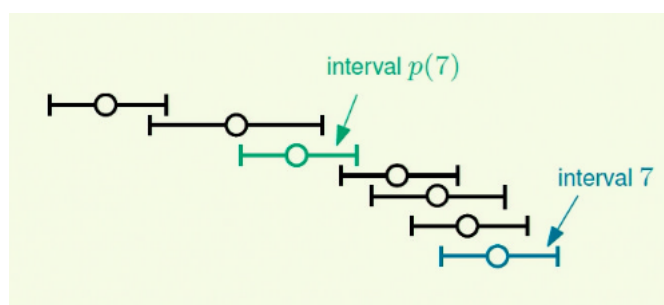
$p(4) = 2$, and $p(2) = 0$ because there is no interval that exists. **Note** that we index from 1 because otherwise 0 would not make sense.

Claim. We can precompute all $p(i)$ in $O(n \log n)$ time. For now, we can assume that this is true, but I will revisit this and prove why. If you did it naively, it would be $O(n^2)$ time. Obviously it's something to do with dynamic programming (because that's what this section is about).

5.1.1 Finding a solution

Consider some optimal schedule $\mathcal{O}$ for intervals $\{1, 2, 3, \ldots, n\}$ with weight OPT. Now, either the nth interval is in schedule $\mathcal{O}$ or it isn't.

Let's look at the case where it is not in $\mathcal{O}$. Schedule $\mathcal{O}$ is also an optimal schedule for the problem with the input consisting of the intervals $\{1, 2, 3, \ldots, n-1\}$ because n is not in it. Therefore, in this case, we have $OPT = OPT(n-1)$. (By the way, $OPT(i)$ is the weight of an optimal schedule for the intervals $\{1, 2, 3, \ldots, i\}$).

Now, let's consider the case that it **is** in $\mathcal{O}$. The only other intervals that could be in $\mathcal{O}$ are $\{1, \ldots, p(n)\}$. We know that everything larger than $p(n)$ is incompatible with the nth schedule.
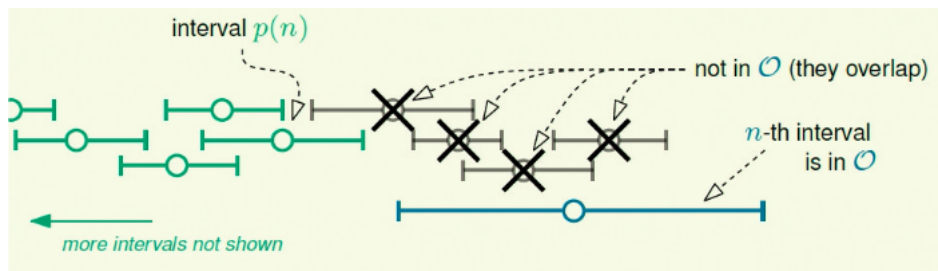
Schedule $\mathcal{O}$ with interval n removed gives an optimal schedule for the intervals $\{1, 2, \ldots, p(n)\}$ so we now have that $OPT = OPT(p(n)) + w_n$.

To summarise:

- **Case 1**: The nth interval is not in $\mathcal{O}$, then $OPT = OPT(n-1)$

- **Case 2**: The nth interval is in $\mathcal{O}$, then $OPT = OPT(p(n)) + w_n$

So, which one do we choose? We choose the bigger one:

$$OPT = \max(OPT(n-1), OPT(p(n)) + w_n)$$

We just now replace all n with i to give us our final recursive algorithm

$$OPT = \max(OPT(i-1), OPT(p(i)) + w_i)$$

There's no easy way to find these solutions, you just have to kind of look at the problem for a while and then it will come to you.

5.1.2 Writing down the recursive algorithm

Now we have the formula, we get the recursive algorithm:

11

```
WIS(i)
    If (i=0)
        Return 0
    Return max(WIS(i-1),WIS(p(i)) + wi)
```

This algorithm is pretty much exponential given the wrong input. If $T(n)$ is the run time of WIS(n) using overlapping intervals, then $T(n) > 2T(n-1)$. This is $O(2^{n/2})$.

5.1.3 Memoization

To solve this, we need to store the solutions to the subproblems.

```
MEMWIS(i)
    If (i=0)
        Return 0
    If WIS[i] undefined
        WIS[i] = max(MEMWIS(i-1),MEMWIS(p(i)) + wi)
    Return WIS[i]
```

In this algorithm, we store the solutions to the previously computed subproblems in an n length array called WIS. The time complexity of computing MEMWIS(n) is now $O(n)$. Unfortunately, linear recursion is still kind of bad.

To compute a value in the array WIS[i], we need both WIS[$i-1$] and WIS[$p[i]$]. Both of these are to the left of WIS[i]. Therefore, we should go through this array from left to right. This is because every time we compute something, we already have all of the things we need.