8.0

Last week we saw dynamic programming algorithms for solving optimization problems.

This week: greedy algorithms.

Notes: Come take a look at your midterm!

Greedy algorithms are similar to DP in that they solve optimization problems by solving a subproblem first. Unlike DP that may depend on many subproblems, greedy algs find one subproblem to move to. Thus, when greedy is guaranteed to give an optimal answer it is more efficient than DP.
In a nutshell, a greedy alg. always makes a choice that seems best at the moment. This approach, as you've already seen, will not always lead to the optimal solution, but when it does it usually gives a fast alg.
As we shall see many problems can be solved very efficiently by greedy algs!

Let us start with an example:

Activity Selection Problem

This is a scheduling problem - we have several activities that have to take place on a single resource. E.g. one lecture hall and all classes of the day. The goal is to fit as many activities.
Modeling this problem we have that the input is a set of activities $\{a_i\}$, each activity has a start time s_i and a finish time f_i s.t. $0 \le s_i \le f_i < \infty$.
If selected, activity a_i starts at time s_i and ends at f_i (and a new activity can start then).
In the activity selection problem we wish to select a maximal subset of activities that are compatible. I.e. that they do not overlap.
For simplicity let us assume that $f_1 \le f_2 \le \dots \le f_n$. (if not then we sort at cost of $O(n \lg n)$).

A natural approach is via DP.

Any solution must have a first activity. If it is a_i then the next activity can start only at time f_i. Thus:

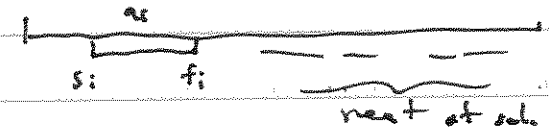
$$opt = 1 + \max_{i \in [f_1, \infty)} opt_{[0, f_i]}$$

(CLRS presents a different DP which is also ok...)

This leads to a simple DP alg.

But let us look more deeply into this.

Assume the optimal is obtained when we first choose a_i.



Then replacing a_i with any a_j, $j < i$ will also work!

Thus, we can actually start by picking the activity that finishes first. We are guaranteed that it is part of some optimal solution.

Then, we are left with a subproblem that can be solved similarly.

```
Greedy-Activity-Selector(s,f)
1. n = s.length
2. A = {a_1}
3. k = 1
4. For m = 2 to n
5.     If s[m] > f[k]
6.         A = A ∪ {a_m}
7.         k = m
8. Return A
```

This alg. runs in linear time (assuming the f_i are sorted).

So what was the process that lead us to the solution?

1. Determine the optimal substructure of the problem (as in DP)

At this stage the problem should be of the form that if we make a choice then we are left with a single subproblem.

2. Prove there is always an optimal solution that locally makes the greedy choice.

3. Show that after making the greedy choice we are left with a subproblem of the same nature.

The greedy choice is what makes greedy algs different from DP algs. We can make a local decision and see how it is part of a global solution whereas in DP we depend on many subproblems and it is not clear which of the options to prefer at each step - first we have to solve several subproblems to make a decision.

Recall we have seen examples where DP does better than greedy (e.g. Matrix chain Multiplication, Rod cutting). Step 2 above is the step that fails in these cases.

8.4.

Let us now discuss an important alg for data compression.

Assume our alphabet contains, say, 32 characters (including spaces, commas etc.).

We can represent each character with 5 bits. Then a document containing 140 characters will be represented by 700 bits.

Can we do better?

Some characters occur more often than others e.g. a, i, e, o, u maybe we can represent them using shorter strings to save in encoding?

This can lead to a problem of decoding. When each char takes 5 bits there is no question when does a char begin and ends. When they have different length it is not clear how to interpret a string.

E.g., if a = 0 e = 00 i = 000 how to interpret 0000000 ?

The problem is that 0 is a char but also the prefix of another char.

The solution is to come up with Prefix-free encoding. I.e. an encoding where no char is a prefix of another char. Then, there can be no ambiguity. Coding schemes for this kind give what is known as prefix codes.

Huffman's encoding scheme is an alg. (a greedy alg.) that constructs an optimal prefix code

What is an optimal prefix code?

Assume we have an alphabet/set of characters C and we know the expected frequency of each symbol $c \in C$ in our file. (we design a different code for each file)

E.g. in Georges Perec's book La Disparition, in French, the letter e only appears ~~once~~ only in the authors name!

8.5

Once we have the frequencies we would like to ~~design~~ match a binary string to each char in a way that minimizes the representation of the file. I.e. one that achieves maximum compression.

Huffman gave a greedy alg. that constructs an optimal prefix code!
His idea is to merge the least frequent symbols to a new symbol ~~with~~ whose frequency is the sum of their frequencies, solve the problem there, and then to "reopen" the merged symbol. I.e. if its string was s, then the original symbols will be matched to $s0, s1$.

This is clearly greedy in a sense that we do a greedy local process — merging two lightest symbols. We have to show that this gives an optimal prefix tree overall.

Showing prefix-free is easy. Optimality requires more work.

Let us assume that symbol $c \in C$ is mapped to a string of length $d(c)$ and it has frequency $c.freq$. Then the length of the document is

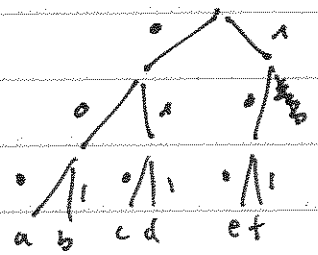
$$\sum_{c \in C} c.freq \cdot d(c)$$

we would like to find a mapping minimizing that.

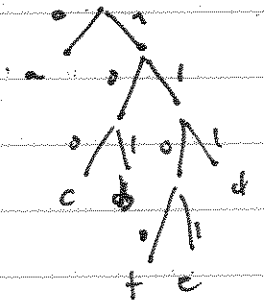
8.6

Before giving Huffman's alg. we note that we can always represent ~~an alphabet~~ a prefix-code with a binary tree.

E.g. a = 000 b = 001 c = 010 d = 011 e = 100 f = 101



a = 0 ~~e~~ = 100 b = 101 d = 111 f = 1100 e = 1101



go In general the root corresponds to the first bit, and a node at ~~leaf~~ depth i to the i^{th} bit.
Since the code is prefix-free all characters are leaves.
Thus, d(c) = depth of c.

Note that an optimal code will always give a full binary tree as otherwise we could save a little. E.g. in 1st example above, better encoding is e = 10 f = 11.

8.7

Having the tree interpretation in mind, Huffman's alg can be written as follows:

```
Huffman(C)
1. n = |C|
2. Q = C                          Q is min-priority-queue keyed by freq. values.
3. For i=1 to n-1
4.     allocate a new node z
5.     z.left = x = Extract-Min(Q)
6.     z.right = y = Extract-Min(Q)
7.     z.freq = x.freq + y.freq
8.     Insert(Q, z)
9. Return Extract-min(Q)
```

At each step we merge two lightest elements and at the end have them as children of the merged element.
Notice that at the end, $|Q| = 1$.

Running-time: We implement Q using binary min-heap.
Line 2 takes $O(n)$ to build the min-heap.
Each iteration of the For loop takes at most $O(\lg n)$
Thus total time is $O(n \lg n)$.

Correctness is based on the following crucial observation.

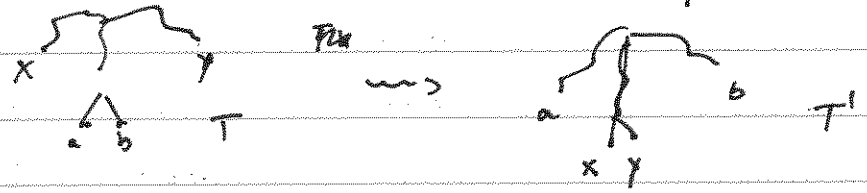
Lemma: Let x, y be with lowest freq. Then, there exists an optimal prefix-code C in which codewords for x and y have the same length and they differ in only ~~one bit~~ the last bit

Pf: We will start with any optimal tree and get another optimal tree in which this property holds.

Let T be an optimal tree and let a, b be siblings at maximal depth

Assume w.l.o.g. $x.freq \le y.freq$, $a.freq \le b.freq$.

We would like to replace x with a and y with b.



what will be the diff between the encodings:

$$\sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) =$$

$$x.freq\left(d_T(x) - d_{T'}(x)\right) + a.freq\left(d_T(a) - d_{T'}(a)\right) + y.freq\left(d_T(y) - d_{T'}(y)\right) + b.freq\left(d_T(b) - d_{T'}(b)\right)$$

Notice $d_T(x) = d_{T'}(a)$ $d_{T'}(x) = d_T(a)$ same for y, b. $\Rightarrow$

$$= (a.freq - x.freq)\left(d_T(a) - d_{T'}(a)\right) + (b.freq - y.freq)\left(d_T(b) - d_{T'}(b)\right)$$

$$= (a.freq - x.freq)\left[d_T(a) - d_T(x)\right] + (b.freq - y.freq)\left(d_T(b) - d_T(y)\right) \ge 0$$

($d_T(a) - d_T(x) \ge 0$ by assumption; $d_T(b) - d_T(y) \ge 0$)

By assumption $x.freq \le y.freq$ are smallest. So it ~~(a,b) ≤ (x,y)~~ then ~~at least one of them is different~~. Thus, the entire sum is non negative.

Thus T' at least as good as T.

The proof also gives that x, y are siblings at maximal depth! ∎

some

We are almost done. We saw that in the optimal code, xy will be siblings at max depth. But ~~is then~~ the question is, do we get an optimal code by our process?

Lemma: Let C' be the new alphabet after "merging" x, y. Let T' be any optimal ~~post~~prefix-code Tree for C'. Let T be obtained by adding two ~~leaves for~~ children to z in the obvious way. Then T is optimal for C.

Proof: Let $B(T'), B(T)$ be the costs of T', T.

Then $B(T) = B(T') + x.freq + y.freq$ (everything stayed the same except z got two new children)

$\Rightarrow B(T') = B(T) - x.freq - y.freq.$

Assume for a contradiction that T is not optimal. Let S be a better tree, where S has x, y as siblings. Let S' be same as S with z instead of x, y.

$$B(S') = B(S) - x.freq - y.freq < B(T) - x.freq - y.freq = B(T')$$

(our assumption)

contradicting T''s optimality. ◎

The two lemmas imply that Huffman is optimal.