lecture 12

interval scheduling

- greedy approach
- weighted intervals and dynamic programming approach

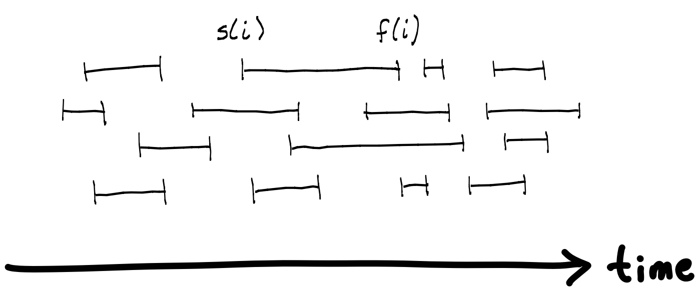
Resources

I used Kleinberg & Tardos to prepare this lecture

chapters 4.1, 6.1, 6.2

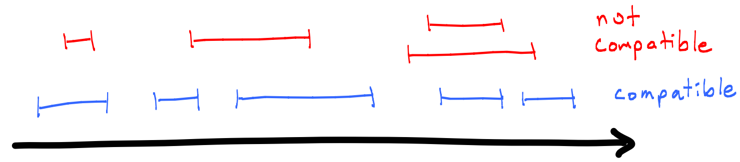
Suppose you have a time period (e.g. a day) and a resource that needs to be shared during that period. It could be a room that is available for booking, or a special instrument such as MRI scanner.

Suppose there are a set of intervals { [s(i), f(i)] } which denote the start and finish times.



Two intervals [s(i), f(i)] and [s(j), f(j)] are **compatible** if they don't overlap.

A set of intervals is compatible if each pair of intervals from that set is compatible.



Exercise:

- Given a set of N intervals, how do you decide if they are compatible ?

<u>Problem 1:</u> given a set of N intervals, choose a compatible subset whose *number* ("cardinality") is as large as possible.

<u>Problem 2:</u> given a set of N intervals, choose a compatible subset whose *total duration* is as long as possible.

(Problem 2 reduces to Problem 1 when all intervals have the same duration.)

Let's look at some "greedy" algorithms for choosing a compatible set of intervals. What is a greedy algorithm?

Kleinberg and Tardos: "... builds up a solution in small steps, choosing a decision at each step *myopically* (short sighted) to optimize some underlying criterion."

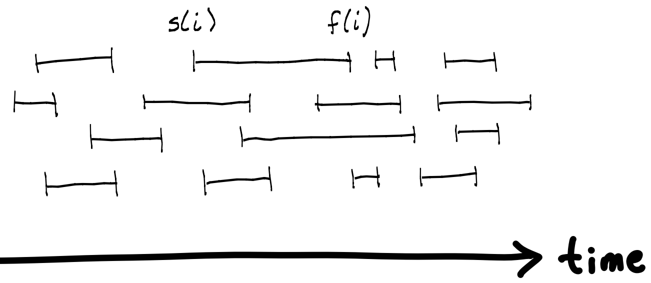
Cormen, Leiserson, Rivest (CLR): ".. makes the choice that looks best in the moment... it makes a locally optimal choice in the hope that the choice will lead to a globally optimal solution".

Levitin: ".. choice must be (1) feasible i.e. satisfy the problem constraints, (2) the best local choice among all feasible choices available at that step, and (3) **irrevocable**".

For example, Dijkstra/Prim/Kruskal's algorithms are all greedy, and they happen to work -- they find a global optimum solution.

Ford-Fulkerson is NOT greedy, since it allows you to undo (reverse) flow to find a better solution.

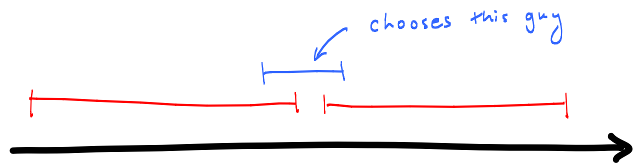
Greedy approaches ?



Greedy approach number 1:

start with an empty set S
repeat {
  choose the **smallest interval** (smallest f(i) - s(i)) that is compatible with all intervals in S, and add this interval to S
} until there are no remaining intervals that are compatible with S

Example where this approaches fails to find the optimum solution for Problem 1 (number of intervals) and Problem 2 (total duration):



Greedy approach 2:

start with an empty set S
repeat {
  choose the interval that has the **smallest value of s(i)**, and that is compatible with all intervals in S, and add it to S
} until there are no remaining intervals that are compatible with S

Example where this approach fails both for problem 1 (maximize the number of intervals) and problem 2 (maximize the total duration of the intervals):

Greedy approach 3:

start with an empty set S
repeat {
  find the interval with the **smallest value of f(i)** and that is compatible with all intervals in S, and add it to S
} until there are no remaining intervals that are compatible with S

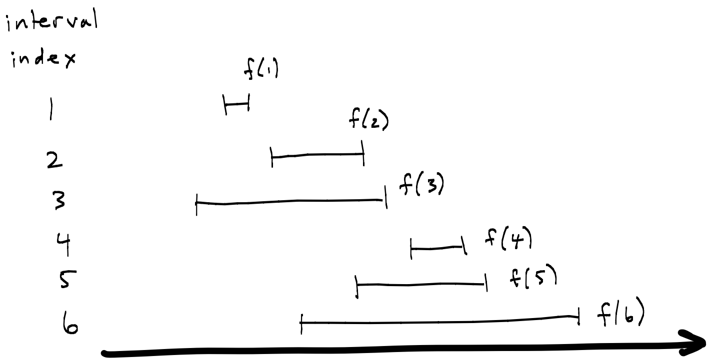
This works for Problem 1 (number of intervals).

Exercises: does it work for Problem 2 also ?

Order the intervals by their finishing time

$f(1) \le f(2) \le f(3) \le \ldots f(N)$

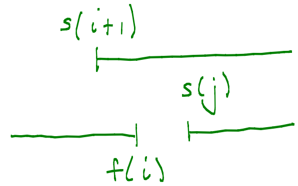
This takes $O(N \log N)$ eg. mergesort.



Greedy approach 3 (Algorithm)

```
// find a maximal set of intervals S

i = 1
S = {1}
for j = 2 to N {
   if s(j) > f(i) {      // compatible ?
      add j to S
      i = j
   }
}
```



Claim: Greedy approach 3 (choose based on earliest finish) finds a maximum compatible solution to problem 1 (most intervals)

Proof:

Assume intervals are ordered by their finishing times: $f(1) \le f(2) \le f(3) \le \cdots f(N)$

Let $i_1, i_2, i_3, \ldots, i_r$ be the indices of solution found by algorithm

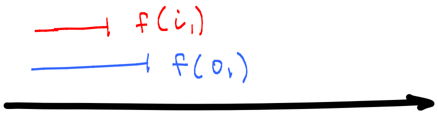
Let $o_1, o_2, o_3, \ldots, o_m$ be the indices of an optimal solution.

We know $r \le m$. Show that $r = m$.

Prove than $f(i_n) \le f(o_n)$ for all $n \le r$.

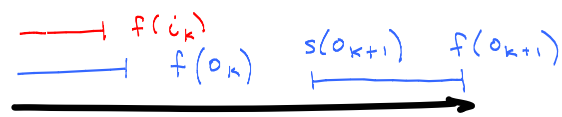
Base case:

$f(i_1) \le f(o_1)$ by definition of algorithm



Induction hypothesis:

$f(i_k) \le f(o_k)$



Induction step

$f(i_k) \le f(o_k) \implies f(i_{k+1}) \le f(o_{k+1})$

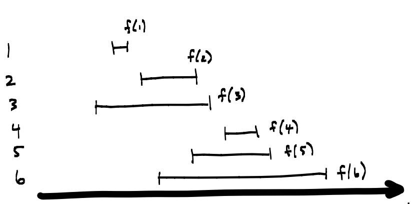
Since algorithm's choice of i_k finishes no later than optimal o_k, the algorithm has at least as many intervals to choose from for i_{k+1}. In particular, it could choose o_{k+1} since $f(i_k) \le f(o_k) < s(o_{k+1})$.

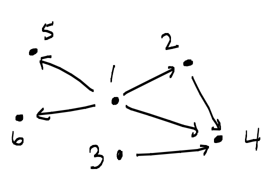
In particular, $f(i_r) \le f(o_r)$.

Next, how do we know $r = m$? If $r < m$ then there would be an interval $[s(o_{r+1}), f(o_{r+1})]$ which is impossible since this interval would be chosen by algorithm too.

ASIDE: Another way to think about the compatibility of intervals:

Define a DAG where vertices are intervals and there is an edge from u to v if $f(u) < s(v)$.





lecture 12

interval scheduling

- greedy approach
- weighted intervals and dynamic programming approach

What is "dynamic programming"?

Term attributed to Bellman (1950's)

CLR: (paraphrase) "Decompose a problem into subproblems. The subproblems are not independent, but rather they share sub-sub problems. The key is to solve each sub-sub problem only once and store these solutions in a table."

We now generalize the interval scheduling problem.

Let interval i have a value v_i.

Choose a set S of compatible intervals that maximizes the sum of values:

$$\text{"Opt"} = \sum_{i \in S} v_i .$$

Claim: earlier problems were special cases.

Problem 1: (number) $v_i \equiv 1$

Problem 2: (total duration) $v_i \equiv f(i) - s(i)$

Greedy 3 won't solve this problem.

We introduce another approach, called "dynamic programming".

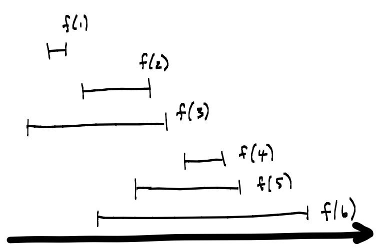
$\{1\}$
$\{1, 2\}$
$\{1, 2, 3\}$
$\{1, 2, 3, 4\}$
$\vdots$
$\{1, 2, 3, 4, \ldots N-1\}$

Consider solving a sequence of smaller versions of the problem.

Again assume interval index is the order of finishing time.

$\{1, 2, 3, 4, \ldots, N\}$ ← original problem

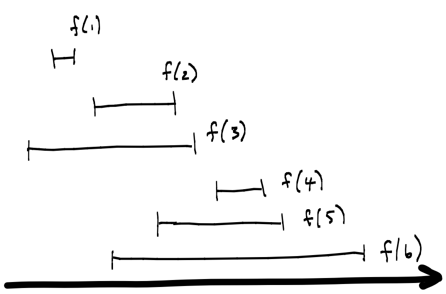
Let $S(i)$ be a set of intervals in maximal solution of the problem when we can use only intervals $\{1, 2, \ldots i\}$.



i	v_i	$S(i)$	$\sum_{S(i)} v_i$
1	4	1	4 = 4
2	8	1, 2	4+8 = 12
3	2	1, 2	4+8 = 12
4	6	1, 2, 4	4+8+6 = 18
5	15	1, 5	4+15 = 19
6	3	1, 5	4+15 = 19

For each interval i, let $p[i]$ be the largest index such that $f(p[i]) < s(i)$.

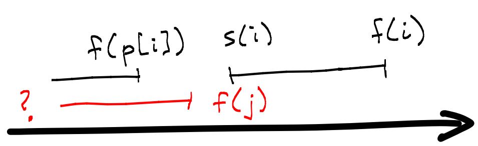
Note: $p[i] < i$



i	$p[i]$
1	0
2	1
3	0
4	3
5	1
6	1

Claim: If $i \in S(i)$ then $S(i)$ cannot contain j where $p[i] < j < i$.

Proof:



To have $p[i] < j < i$, we would need $f(p[i]) < f(j)$ and $f(j) < s(i)$.

But this would contradict the definition of $p[i]$.

Let the maximum value using intervals $\{1, 2, \ldots i\}$ be

$$Opt(i) \equiv \sum_{j \in S(i)} v_j .$$

Claim:

$$Opt(i) = \max \{ Opt(i-1),\ v_i + Opt(p[i]) \}$$

$S(i)$ does not contain i.

$S(i)$ contains i.

"Dynamic Programming" Algorithm

Define array $Opt[0, \ldots N]$

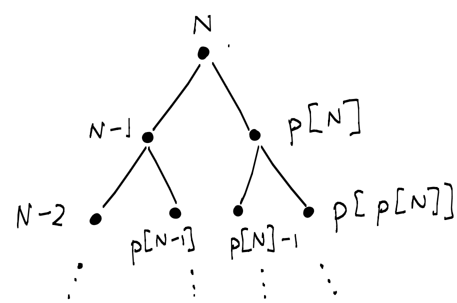
```
Opt[0] = 0
for i = 1 to N
    Opt[i] = max{ Opt[i-1],
                  v_i + Opt[p[i]] }
return Opt[N]
```

Note: Assuming $p[\ ]$ has been pre-computed*, this algorithm takes $O(N)$.

What about a recursive algorithm?

```
ComputeOpt(i) {
    if n == 0
        return 0
    else
        return max { computeOpt(i-1),
                     v_i + computeOpt(p[i]) }
}
```

Call tree for (recursive) computeOpt



For large N, this can blow up.

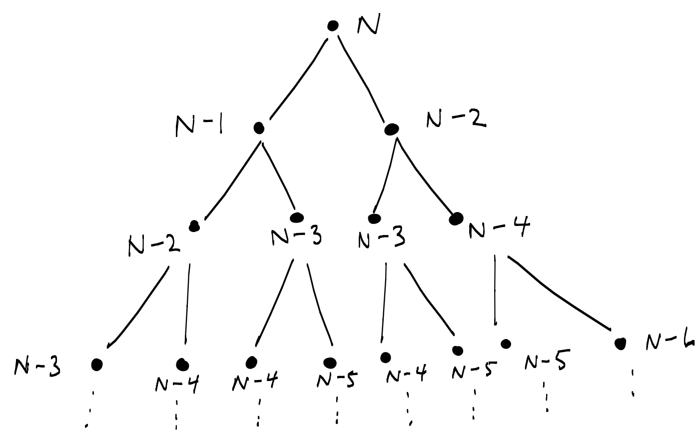
Analogy: Fibonacci

$F(0) = 0$
$F(1) = 1$
$F(n+1) = F(n) + F(n-1)$

n	0	1	2	3	4	5	6	7
$F(n)$	0	1	1	2	3	5	8	13

Suppose you try to compute $F(N)$ using recursion, where N is large.

Call Tree for recursive Fibonacci



Note the redundant computation!

We can use recursion for Fibonacci but we must be careful to avoid redundant computation.

Use a global array F[0,... N]

F[0] = 0, F[1] = 1
F[i] = −1 for all i = 2, ... N

```
Fibonacci(k) {
   if   k == 0  or  k == 1    return F[k]
   else {  if  F[k-1] < 0
                F[k-1] = fibonacci(k-1)
           if  F[k-2] < 0
                F[k-2] = fibonacci(k-2)
           return  F[k-1] + F[k-2]
        }
}
```

"Memo-ization"

Save values that have already been computed so that you dont have to compute them again.

How do we apply this to our weighted interval scheduling problem?

Notation: $Opt[i] \equiv \sum_{v_j \in S(i)} v_j$

$Opt[0] = 0$
for all $i \in \{1, 2, \dots, N\}$
$Opt[i] = -1$ // stores maximum values

```
computeOpt(n) {
   if  n == 0,  return 0
   else  if Opt[n] ≥ 0,  return Opt[n]
   else {Opt[n] = max{ computeOpt(n-1),
                       v_n + computeOpt(p[n]) }}
   return Opt[n]
}
```

What is O( ) running time?

Each call to computeOpt( ) either performs a constant number of operations and then returns, or else sets one value of Opt[ ] and performs two calls to computeOpt( )).

Since each value of Opt[ ] is set only once, the time required is $O(N)$, same as iterative solution.

[Don't forget about $O(N \log N)$ needed to sort the intervals by finishing time.]

Q: Is this a greedy algorithm?

A: No. Although Opt[i] increases as i increases, the set S(i) does not necessarily grow as i increases.

i	v_i	$S(i)$	$\sum_{S(i)} v_i$
1	4	1	4
2	8	1, 2	12
3	2	1, 2	12
4	6	1, 2, 4	18
5	15	1, 5	19
6	3	1, 5	19

We could compute S(n) while Opt[ ] is being computed, or do it afterwards as follows.

```
findS(n) {
  if  n > 0  {
      if  Opt[n-1] == Opt[n]
          //  S(n) doesn't contain n.
          return   findS(n-1)
      else {
          return  {n} ∪ findS(p[n])
      }
  else  return {}    // the empty set
}
```

my apologies for the notation – curly brackets used in two different ways