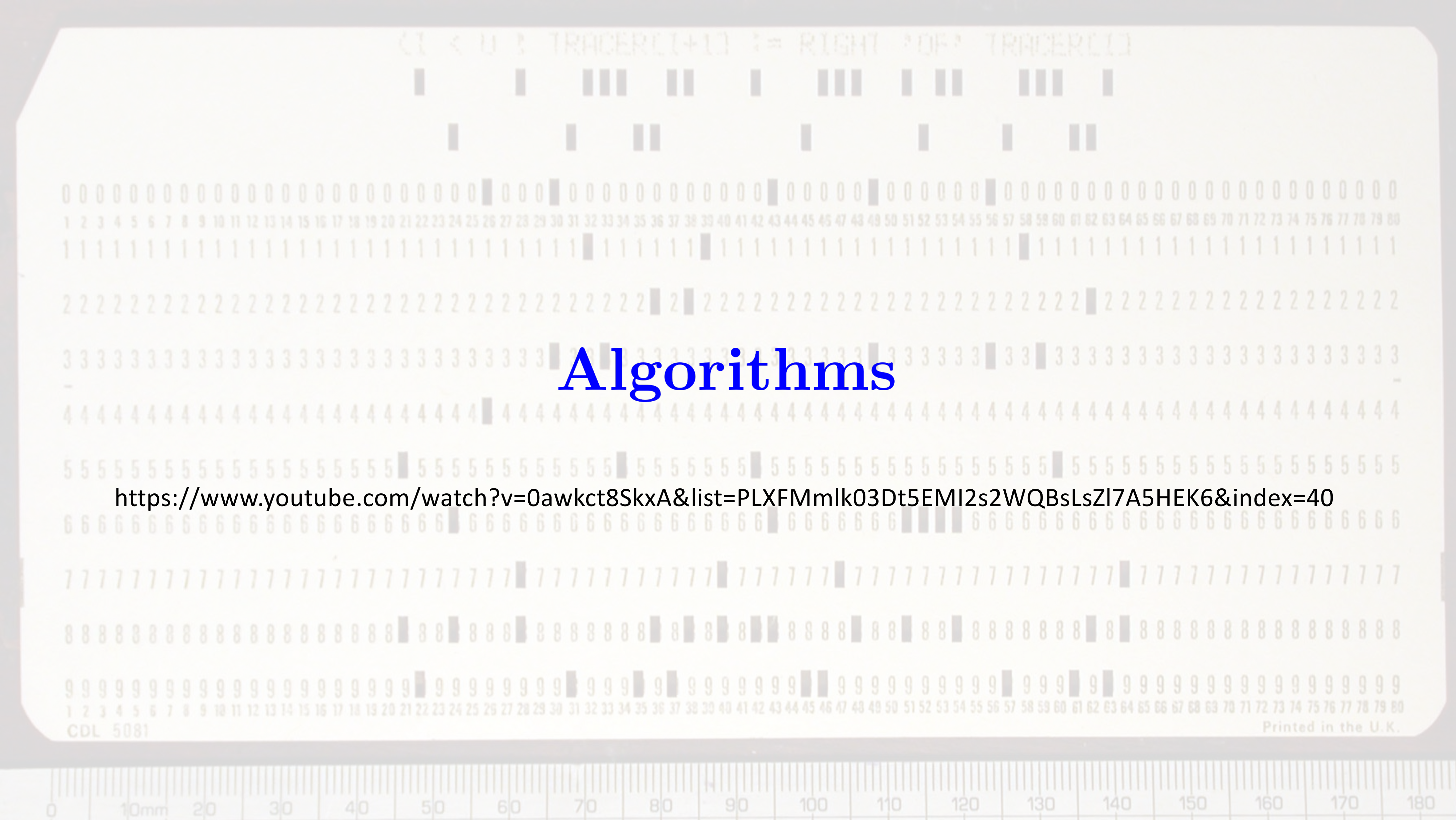


Algorithms

https://www.youtube.com/watch?v=0awkct8SkxA&list=PLXFMmlk03Dt5EMI2s2WQBsLsZl7A5HEK6&index=40

Dynamic Programming

Consider the following code for computing fibonnaci numbers:

```
def fib(n):
  if n < 2:
     return 1
  else:
     return fib(n-1) + fib(n-2)
```

Recurrence relation for running time:

$$T(n) = T(n-1) + T(n-2) + O(1), T(1) = O(1)$$

$T(n)$ is exponential in n.

Traditional divide and conquer does not work

Observation. We compute `fib`(k) for the same k several times!

Why not just save it when we compute it?

Dynamic Programming

Modified Algorithm.

```
M = [1, 1, ∞, ···, ∞]

def fib(n):

    if n < 2:  return 1

    if M[n-1] == ∞:  M[n-1] = fib(n-1)

    if M[n-2] == ∞:  M[n-2] = fib(n-2)

    M[n] = M[n-1] + M[n-2]

    return M[n]
```

array of size $n+1$ where $M[0] = M[1] = 1$, and for $i = 2..n$, $M[i] = \infty$ indicating that `fib`*(i) has not yet been computed.*

$O(n)$ time algorithm

Takes more space but drastically reduces the running time

The idea of saving results is called ***memoization***.

A similar idea can be used in many problems.

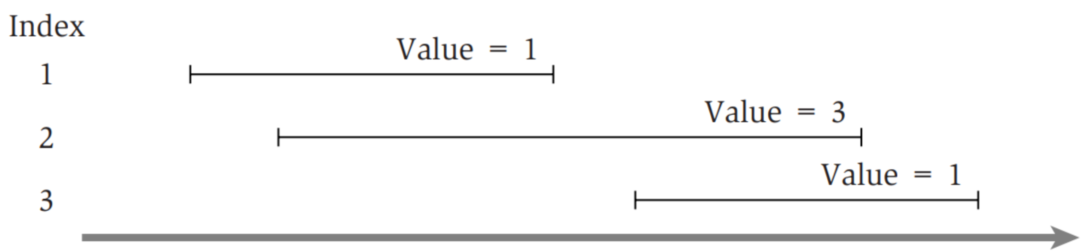
Weighted Interval Scheduling

We are given n jobs numbered $1..n$.

The i^{th} job starts at time s_i, finishes at time t_i and has value v_i.

Goal. Pick a subset of non-overlapping jobs of maximum total value.

Example.



Does our earlier greedy algorithm of always picking the job with the earliest finish time work?

No!

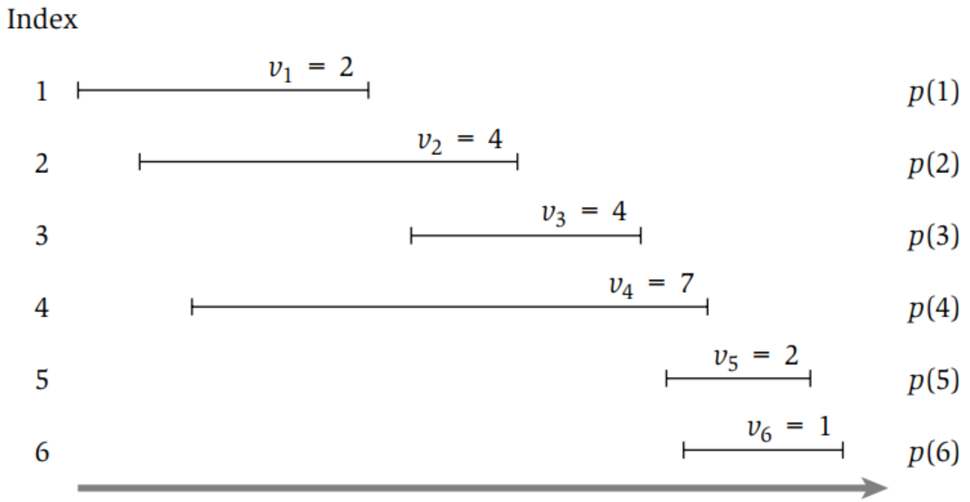
Weighted Interval Scheduling

By sorting, we assume that $f_1 \le f_2 \le \cdots \le f_n$. *sorted in non-decreasing order of finish times*

We also define a dummy job 0 s.t $f_0 < s_1$.

For any $j \in 1..n$, define $p(j) = \max\{k \in 0..n : f_k < s_j\}$. *job with the maximum finish time before the j's start time*

Example.



Compute the values

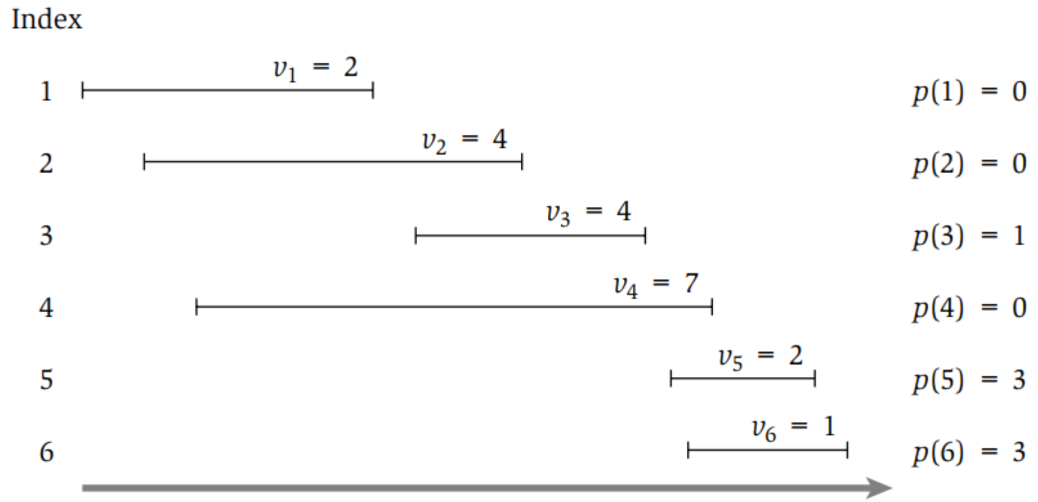
Weighted Interval Scheduling

By sorting, we assume that $f_1 \leq f_2 \leq \cdots \leq f_n$. *sorted in non-decreasing order of finish times*

We also define a dummy job 0 s.t $\underline{f_0 < s_1}$. **should be $<\boldsymbol{s_i}$ for all $\boldsymbol{i} \in \{1, \ldots, \boldsymbol{n}\}$ so that all $\boldsymbol{p(i)}, \boldsymbol{i} \in \{1, .., \boldsymbol{n}\}$ are defined**

For any $j \in 1..n$, define $p(j) = \max\{k \in 0..n : f_k < s_j\}$. *job with the maximum finish time before the j's start time*

Example.



How fast can we compute $p(j)$ for all $j = 1 \,..\, n$? $O(n \log n)$ *time*

Weighted Interval Scheduling

Let $\textsc{Opt}(j)$ define the value of the optimal solution among jobs $1 \mathinner{..} j$.

Consider the last job j. We have two options for job j: take it or leave it.

If we take it, we need to find the optimal solution among jobs $1 \mathinner{..} p(j)$.

If we leave it, we need to find the optimal solution among jobs $1 \mathinner{..} (j-1)$.

Thus, $\textsc{Opt}(j) = \max \{ v_j + \textsc{Opt}(p(j)), \textsc{Opt}(j-1) \}$.

```
def ComputeOpt(j):
  if j==1:  return v1
  return max( vj + ComputeOpt(p(j)), ComputeOpt(j-1) )
```

We assume that $p(j)$ has already been computed for each j.

How much time does this take?

Weighted Interval Scheduling

```
def ComputeOpt(j):
  if j==1:  return v_1
  return max( v_j + ComputeOpt(p(j)), ComputeOpt(j-1) )
```

Let $T(j)$ denote the time required by `computeOpt(j)`

Recurrence relation:

$$T(j) = T(p(j)) + T(j-1) + O(1),\, T(1) = O(1).$$

In the worst case, we could have $p(j) = j - 1$ for all $j = 1 \mathinner{..} n$.

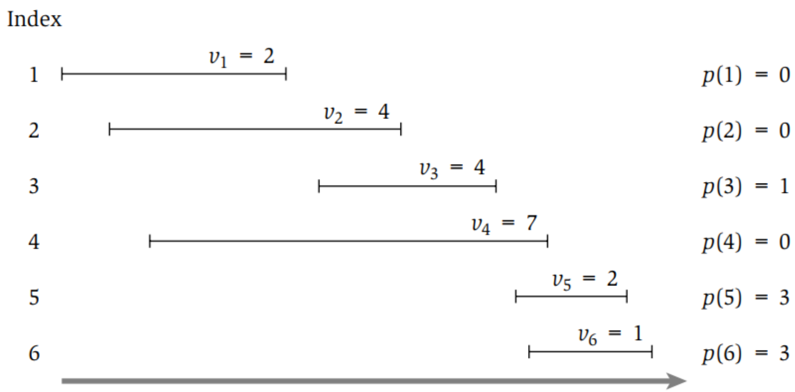
Then, $T(j) = 2\,T(j-1) + O(1)$, $T(1) = O(1)$ for all $j = 1 \mathinner{..} n$

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

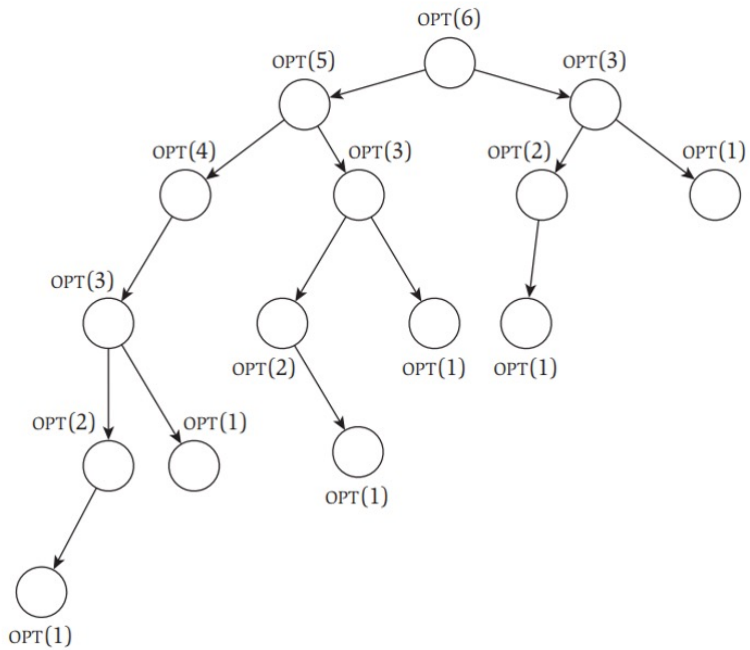
Weighted Interval Scheduling

Example.

Input



Recursion Tree



Observation. Many subproblems are computed multiple times!

How do we improve the running time? *Memoization!*

Weighted Interval Scheduling

```
M = [0, ∞, ···, ∞]
```

array of length $(n+1)$ where $M[0] = 0$, and $M[1] = M[2] = \cdots = M[n] = \infty$.

```
def ComputeOpt(j):
    if j==1:  return v_1
    if M[p(j)] == ∞: M[p(j)] = ComputeOpt(p(j))
    if M[j-1] == ∞:  M[j-1] = ComputeOpt(j-1)
    M[j] = max(v_j + M[p(j)], M[j-1])
    return M[j]
```

we assume $j \in 1\,..\,n$

Assumption: $p(j)$ has already been computed for each j.

Running time for `ComputeOpt(n)`? *$O(n)$* *Why?*

For any $j \in 1..n$, `ComputeOpt(j)` is called at most once, since after any such call, the value is memoized and stored in $M[j]$.

For any j, the time spent in `ComputeOpt(j)` apart from recursive calls is $O(1)$.

Thus the total time is $O(n)$. *excluding the $O(n \log n)$ time for computing the $p(j)$'s*

Weighted Interval Scheduling

We have computed the value of the optimal solution. How do we find the jobs in the optimal solution?

```
M = array computed before
S = []   stores selected jobs
j = n
while j>0:
    if M[j] == v_j + M[p(j)]: ← indicates if the j^th job belongs to OPT(j)
        S.append(j)
        j = p(j)
    else:  j = j-1
```

How much time does this take? $O(n)$

EXERCISE

Subset Sum

Given an array $A[1..n]$ containing positive numbers we need to check if a subset of the numbers in the array sum to a given number T.

Example. 1 2 5 7

Is there a subset that sums to 13*?* *Yes.*

Is there a subset that sums to 0*?* *Yes, the empty subset!*

Is there a subset that sums to 4*?* *No.*

How fast can we solve this problem?

Solution

Subset Sum

Let $SS(i,t) = \texttt{TRUE}$ iff some subset of $A[i..n]$ sums to t.

We seek $SS(1,T)$.

$$SS(i,t) = \begin{cases} \texttt{TRUE} & \text{if } t = 0 \\ \texttt{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } A[i] > t \\ SS(i+1,t) \ \textbf{OR} \ SS(i+1, t - A[i]) & \text{otherwise} \end{cases}$$

In what order should we do the bottom up computation?

In decreasing order of i and increasing order of t.

For what range of values of i and t do we need to memoize $SS(i,t)$?

For $i = 1$ to n and for $t = 1$ to T.

Running time? *$O(nT)$.*

EXERCISE

Longest Increasing Subsequence

Exercise.

Suppose that we are given an array A with n numbers and we want to compute the length of the longest increasing subsequence in A.

Example. $1 \quad 4 \quad 2 \quad 8 \quad 5 \quad 7$

There are several increasing subsequences: $1 \quad 2 \quad 8$, $4 \quad 8$, $1 \quad 4 \quad 7$ *etc.*

The longest ones among these are $1 \quad 2 \quad 5 \quad 7$ *and* $1 \quad 4 \quad 5 \quad 7$*.*

How fast can we compute the longest increasing subsequence?