Dynamic Programming I

Algorithm Design & Analysis

Spring 2019
Outline

Memoizing: A Motivating Example

Weighted Interval Scheduling: A First Dynamic Programming Example

Multi-Parameter Dynamic Programming Problems
  Some Simple Examples: Subset Sum & Knapsack
Announcements

Clarification: For solving recurrences, you may assume that \( n = b^k \) (\( b = 2 \), say) for simplicity. For proving algorithm correctness, you should account for all values of \( n \).
Fibonacci Numbers

How long does it take to compute the $n^{th}$ Fibonacci number?
**Fibonacci Numbers**

How long does it take to compute the \(n^{th}\) Fibonacci number?

\[
\text{fib}(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
\text{fib}(n - 1) + \text{fib}(n - 2) & \text{otherwise}
\end{cases}
\]
Fibonacci Numbers

How long does it take to compute the $n^{th}$ Fibonacci number?

$$fib(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
fib(n - 1) + fib(n - 2) & \text{otherwise} 
\end{cases}$$

Using recursion, number of calls to $fib()$ made by $fib(n)$ is 

Well-known fact: 

$$fib(n) \geq \left\lceil \frac{1 + \sqrt{5}}{2} \right\rceil^n \geq \frac{1}{\sqrt{5}} \cdot 2^{n-2}$$

So $C(n) \geq 1.6^n$ for all $n \geq 1$.

That is, $C(n)$ grows exponentially!
Fibonacci Numbers

How long does it take to compute the $n^{th}$ Fibonacci number?

$$fib(n) = \begin{cases} 1 & \text{if } n = 1, 2 \\ fib(n - 1) + fib(n - 2) & \text{otherwise} \end{cases}$$

Using recursion, number of calls to $fib()$ made by $fib(n)$ is

$$C(n) = \begin{cases} 1 & \text{if } n = 1, 2 \\ 1 + C(n - 1) + C(n - 2) & \text{otherwise} \end{cases}$$
Fibonacci Numbers

How long does it take to compute the $n^{th}$ Fibonacci number?

$$fib(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
fib(n - 1) + fib(n - 2) & \text{otherwise}
\end{cases}$$

Using recursion, number of calls to $fib()$ made by $fib(n)$ is

$$C(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
1 + C(n - 1) + C(n - 2) & \text{otherwise}
\end{cases}$$

Thus $C(n) \geq fib(n)$ for all $n \geq 1$ (simple induction proof)
Fibonacci Numbers

How long does it take to compute the $n^{th}$ Fibonacci number?

$$fib(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
fib(n - 1) + fib(n - 2) & \text{otherwise}
\end{cases}$$

Using recursion, number of calls to $fib()$ made by $fib(n)$ is

$$C(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
1 + C(n - 1) + C(n - 2) & \text{otherwise}
\end{cases}$$

Thus $C(n) \geq fib(n)$ for all $n \geq 1$ (simple induction proof)

**Well-known fact:** $fib(n) \geq \left(\frac{1 + \sqrt{5}}{2}\right)^{n-2} \geq 1.6^{n-2}$
**Fibonacci Numbers**

How long does it take to compute the $n^{th}$ Fibonacci number?

$$fib(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
fib(n - 1) + fib(n - 2) & \text{otherwise}
\end{cases}$$

Using recursion, number of calls to $fib()$ made by $fib(n)$ is

$$C(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
1 + C(n - 1) + C(n - 2) & \text{otherwise}
\end{cases}$$

Thus $C(n) \geq fib(n)$ for all $n \geq 1$ (simple induction proof)

**Well-known fact:** $fib(n) \geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \geq 1.6^{n-2}$

So $C(n) \geq 1.6^{n-2}$ for all $n \geq 1$
Fibonacci Numbers

How long does it take to compute the \( n^{th} \) Fibonacci number?

\[
fib(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
n - 1 + fib(n - 1) + fib(n - 2) & \text{otherwise}
\end{cases}
\]

Using recursion, number of calls to \( fib() \) made by \( fib(n) \) is

\[
C(n) = \begin{cases} 
1 & \text{if } n = 1, 2 \\
1 + C(n - 1) + C(n - 2) & \text{otherwise}
\end{cases}
\]

Thus \( C(n) \geq fib(n) \) for all \( n \geq 1 \) (simple induction proof)

**Well-known fact:** \( fib(n) \geq \left( \frac{1 + \sqrt{5}}{2} \right)^{n-2} \geq 1.6^{n-2} \)

So \( C(n) \geq 1.6^{n-2} \) for all \( n \geq 1 \)

That is, \( C(n) \) grows exponentially!
Iterative Fibonacci

Algorithm 1 Fibonacci

procedure FIB(n)
    if $n \leq 2$ then return 1
    $F_n = F_c = 1$
    for $i \leftarrow 3$ to $n$ do
        $t = F_n$
        $F_n = F_n + F_c$
        $F_c = t$
    return $F_n$
end procedure
Iterative Fibonacci

Algorithm 2 Fibonacci

procedure FIB(n)
    if \( n \leq 2 \) then return 1
    \( F_n = F_c = 1 \)
    for \( i \leftarrow 3 \) to \( n \) do
        \( t = F_n \)
        \( F_n = F_n + F_c \)
        \( F_c = t \)
    return \( F_n \)
end procedure

But what if we are computing many Fibonacci numbers for repeated use in a program?
Algorithm 3: Fibonacci Table

procedure FIBTABLE(n)
    \[ \text{for } i \leftarrow 3 \text{ to } i \leftarrow n \text{ do} \]
    \[ F[i] = F[i - 1] + F[i - 2] \]
end procedure
Algorithm 4 Fibonacci Table

procedure {\texttt{FIBTABLE}}(n)
\begin{align*}
\text{for } i &\leftarrow 3 \text{ to } i \leftarrow n \text{ do} \\
F[i] &= F[i-1] + F[i-2]
\end{align*}
end procedure

What if we don’t know in advance which Fibonacci numbers we’ll need?
**Algorithm 5** Fibonacci Table

```plaintext
procedure FIBTABLE(n)
    for i ← 3 to i ← n do
        F[i] = F[i - 1] + F[i - 2]
    end procedure
```

What if we don’t know in advance which Fibonacci numbers we’ll need?

Memoization: Fill table opportunistically....
Recursive Fibonacci with Memoizing

Algorithm 6 Fibonacci with Memoizing

procedure MEMOFIB(F, n) // Prior to first call, F[1..n] has been set to 0
    if F[n] > 0 then
        return F[n]
    else if n = 1, 2 then
        F[n] = 1
        return F[n]
    else
        F[n] = memoFib(F, n - 1) + memoFib(F, n - 2)
        return F[n]
end procedure

Memoizing is very useful for making recursion more efficient!
**Recursive Fibonacci with Memoizing**

**Algorithm 7** Fibonacci with Memoizing

```plaintext
procedure MEMOFib(F, n)// Prior to first call, F[1..n] has been set to 0
    if F[n] > 0 then
        return F[n]
    else if n = 1, 2 then
        F[n] = 1
        return F[n]
    else
        F[n] = memoFib(F, n − 1) + memoFib(F, n − 2)
        return F[n]
end procedure
```

Memoizing is very useful for making recursion more efficient!
Weighted Interval Scheduling

**The Input:** Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).
**Weighted Interval Scheduling**

**The Input:** Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).

**The Output:** A subset \(I \subseteq \{1, \ldots, n\}\), where the intervals \(\{(s_i, t_i) : i \in I\}\) are pairwise non-intersecting intervals that maximize \(\sum_{i \in I} v_i\).
Weighted Interval Scheduling

The Input: Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).

The Output: A subset \(I \subseteq \{1, \ldots, n\}\), where the intervals \(\{(s_i, t_i) : i \in I\}\) are pairwise non-intersecting intervals that maximize \(\sum_{i \in I} v_i\).

This notation is nicer than, say, \((s_{i_1}, t_{i_1}), \ldots, (s_{i_k}, t_{i_k})\) and \(\sum_{j=1}^k v_{i_j}\)
**Weighted Interval Scheduling**

**The Input:** Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).

**The Output:** A subset \(I \subseteq \{1, \ldots, n\}\), where the intervals \(\{(s_i, t_i) : i \in I\}\) are pairwise non-intersecting intervals that maximize \(\sum_{i \in I} v_i\).

This notation is nicer than, say, \((s_{i_1}, t_{i_1}), \ldots, (s_{i_k}, t_{i_k})\) and \(\sum_{j=1}^k v_{i_j}\).

Let’s simplify: Can we find the value of the best solution, not the actual set of intervals. That is, find the largest \(\sum_{i \in I} v_i\) where the intervals in \(I\) are compatible.
**Weighted Interval Scheduling**

**The Input:** Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).

**The Output:** A subset \(I \subseteq \{1, \ldots, n\}\), where the intervals \(\{(s_i, t_i) : i \in I\}\) are pairwise non-intersecting intervals that maximize \(\sum_{i \in I} v_i\).

This notation is nicer than, say, \((s_{i_1}, t_{i_1}), \ldots, (s_{i_k}, t_{i_k})\) and \(\sum_{j=1}^k v_{i_j}\)

Let's simplify: Can we find the value of the best solution, not the actual set of intervals. That is, find the largest \(\sum_{i \in I} v_i\) where the intervals in \(I\) are compatible.

Let \(\text{maxSched}(n)\) be the value of the optimal schedule.
Weighted Interval Scheduling

The Input: Given intervals \((s_1, t_1), \ldots, (s_n, t_n)\) where each \((s_i, t_i)\) has non-negative value (weight) \(v_i\).

The Output: A subset \(I \subseteq \{1, \ldots, n\}\), where the intervals \\{\((s_i, t_i) : i \in I\)\} are pairwise non-intersecting intervals that maximize \(\sum_{i \in I} v_i\).

This notation is nicer than, say, \((s_{i_1}, t_{i_1}), \ldots, (s_{i_k}, t_{i_k})\) and \(\sum_{j=1}^{k} v_{i_j}\)

Let’s simplify: Can we find the value of the best solution, not the actual set of intervals. That is, find the largest \(\sum_{i \in I} v_i\) where the intervals in \(I\) are compatible.

Let \(\text{maxSched}(n)\) be the value of the optimal schedule

Can we find a recurrence for \(\text{maxSched}(n)\)?
Weighted Interval Scheduling

Observations
Weighted Interval Scheduling

Observations

• Assume the intervals are sorted by increasing $t$-value.
Weighted Interval Scheduling

Observations

• Assume the intervals are sorted by increasing $t$-value.
• For $i \leq n$, let $maxSched(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
Weighted Interval Scheduling

Observations

• Assume the intervals are sorted by increasing t-value.
• For $i \leq n$, let $maxSched(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
• If $(s_n, t_n)$ isn’t used: $maxSched(n) = maxSched(n - 1)$
Weighted Interval Scheduling

Observations

- Assume the intervals are sorted by increasing $t$-value.
- For $i \leq n$, let $maxSched(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
- If $(s_n, t_n)$ isn’t used: $maxSched(n) = maxSched(n - 1)$

But what if it is?
Weighted Interval Scheduling

Observations

- Assume the intervals are sorted by increasing \( t \)-value.
- For \( i \leq n \), let \( \text{maxSched}(i) \) be value of the optimal schedule using only intervals in \( \{1, \ldots, i\} \)
- If \( (s_n, t_n) \) isn’t used: \( \text{maxSched}(n) = \text{maxSched}(n - 1) \)

But what if it is?

- If interval \( (s_n, t_n) \) is used, then no interval \( (s_j, t_j) \) with \( j < n \) and \( s_n \leq t_j \leq t_n \) is used (overlapping!)
Weighted Interval Scheduling

Observations

- Assume the intervals are sorted by increasing $t$-value.
- For $i \leq n$, let $maxSched(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
- If $(s_n, t_n)$ isn’t used: $maxSched(n) = maxSched(n - 1)$

But what if it is?

- If interval $(s_n, t_n)$ is used, then no interval $(s_j, t_j)$ with $j < n$ and $s_n \leq t_j \leq t_n$ is used (overlapping!)
- So let $p_n$ be the interval with the latest finish time that doesn’t intersect interval $(s_n, t_n)$
Weighted Interval Scheduling

Observations

- Assume the intervals are sorted by increasing $t$-value.
- For $i \leq n$, let $\text{maxSched}(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
- If $(s_n, t_n)$ isn’t used: $\text{maxSched}(n) = \text{maxSched}(n - 1)$

But what if it is?

- If interval $(s_n, t_n)$ is used, then no interval $(s_j, t_j)$ with $j < n$ and $s_n \leq t_j \leq t_n$ is used (overlapping!)
- So let $p_n$ be the interval with the latest finish time that doesn’t intersect interval $(s_n, t_n)$
- So only intervals $(s_j, t_j)$ with $j \leq p_n$ can be used with $(s_n, t_n)$
Weighted Interval Scheduling

Observations

• Assume the intervals are sorted by increasing $t$-value.
• For $i \leq n$, let $\text{maxSched}(i)$ be value of the optimal schedule using only intervals in $\{1, \ldots, i\}$
• If $(s_n, t_n)$ isn’t used: $\text{maxSched}(n) = \text{maxSched}(n - 1)$

But what if it is?

• If interval $(s_n, t_n)$ is used, then no interval $(s_j, t_j)$ with $j < n$ and $s_n \leq t_j \leq t_n$ is used (overlapping!)
• So let $p_n$ be the interval with the latest finish time that doesn’t intersect interval $(s_n, t_n)$
• So only intervals $(s_j, t_j)$ with $j \leq p_n$ can be used with $(s_n, t_n)$
• So if $(s_n, t_n)$ is used: $\text{maxSched}(n) = v_n + \text{maxSched}(p_n)$
Weighted Interval Scheduling

This gives

\[ \text{maxSched}(n) = \max\{ \text{maxSched}(n - 1), v_n + \text{maxSched}(p_n) \} \]
Weighted Interval Scheduling

This gives

\[ \text{maxSched}(n) = \max\{ \text{maxSched}(n-1), v_n + \text{maxSched}(p_n) \} \]

To generalize: Store, for each \( i \), largest \( j < i \) such that \( t_j < s_i \)
in a table \( p[] \): \( p[i] = j \) (for predecessor)
Weighted Interval Scheduling

This gives

\[ \text{maxSched}(n) = \max\{ \text{maxSched}(n - 1), v_n + \text{maxSched}(p_n) \} \]

To generalize: Store, for each \( i \), largest \( j < i \) such that \( t_j < s_i \) in a table \( p[] \): \( p[i] = j \) (for predecessor)

Then we can write
Weighted Interval Scheduling

This gives

$$maxSched(n) = \max\{maxSched(n-1), v_n + maxSched(p_n)\}$$

To generalize: Store, for each $i$, largest $j < i$ such that $t_j < s_i$ in a table $p[]$: $p[i] = j$ (for predecessor)

Then we can write

$$maxSched(i) = \max\{maxSched(i-1), v_i + maxSched(p(i))\}$$
The Algorithm
The Algorithm

**Algorithm 9** MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: \( p[1..n] \) has been constructed
Require: And a table \( M[1..n] \) has been initialized to 0

procedure maxSched(n)
    if n = 0 then
        return 0
    else if \( M[n] > 0 \) then
        return \( M[n] \)
    else
        \( M[n] = \max\{ \maxSched(n-1), v[n] + \maxSched(p[n]) \} \)
    return \( M[n] \)
end procedure
The Algorithm

Algorithm 10 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
The Algorithm

Algorithm 11 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: $p[1..n]$ has been constructed
Require: And a table $M[1..n]$ has been initialized to 0
The Algorithm

Algorithm 12 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: $p[1..n]$ has been constructed
Require: And a table $M[1..n]$ has been initialized to 0

procedure MAXSCHED($n$)
The Algorithm

Algorithm 13 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: \( p[1..n] \) has been constructed
Require: And a table \( M[1..n] \) has been initialized to 0

procedure \text{MAXSCHED}(n)
    if \( n = 0 \) then
        return 0

else if \( M[n] > 0 \) then
    return \( M[n] \)
else
    \( M[n] = \max\{\text{MAXSCHED}(n-1), v_n + \text{MAXSCHED}(p[n])\} \)

return \( M[n] \)
The Algorithm

Algorithm 14 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: $p[1..n]$ has been constructed
Require: And a table $M[1..n]$ has been initialized to 0

procedure MAXSCHED($n$)

if $n = 0$ then
    return 0
else if $M[n] > 0$ then
    return $M[n]$
The Algorithm

Algorithm 15 MaxSched with Memoizing

Require: Intervals have been sorted by increasing finish time
Require: $p[1..n]$ has been constructed
Require: And a table $M[1..n]$ has been initialized to 0

procedure MAXSCHED($n$)

if $n = 0$ then
    return 0
else if $M[n] > 0$ then
    return $M[n]$
else
    $M[n] = \max\{\text{maxSched}(n - 1), v_n + \text{maxSched}(p[n])\}$
    return $M[n]$
end procedure
Iterative Weighted Interval Scheduling

Algorithm 16 Iterated MaxSched

\begin{procedure}
\textsc{MaxSched}(n)
\end{procedure}
Algorithm 17 Iterated MaxSched

procedure MAXSCHED\(n\)

Require: Intervals have been sorted by increasing finish time

Require: Prior to first call, \(p[1..n]\) has been constructed
Iterative Weighted Interval Scheduling

Algorithm 18 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time
Require: Prior to first call, $p[1..n]$ has been constructed

$M[0] = 0$
Iterative Weighted Interval Scheduling

Algorithm 19 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time
Require: Prior to first call, \( p[1..n] \) has been constructed

\[
M[0] = 0
\]

for \( i \leftarrow 1 \) to \( i \leftarrow n \) do

\[
M[i] = \max\{M[i - 1], v_i + M[p[i]]\}
\]

end procedure

Notes

- \( \text{Runtime is } O(n) + \text{time to sort intervals + time to build } p[] \)
- Claim: \( p[] \) can be constructed in \( O(n \log n) \) time (we’ll come back to this)
- Thus time complexity is \( O(n \log n) \); space complexity is \( O(n) \)
Iterative Weighted Interval Scheduling

Algorithm 20 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time
Require: Prior to first call, p[1..n] has been constructed

\[ M[0] = 0 \]

for \( i \leftarrow 1 \) to \( i \leftarrow n \) do

\[ M[i] = \max\{ M[i - 1], v_i + M[p[i]] \} \]

end procedure
Iterative Weighted Interval Scheduling

Algorithm 21 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time

Require: Prior to first call, p[1..n] has been constructed

M[0] = 0

for i ← 1 to i ← n do

M[i] = max{M[i − 1], vi + M[p[i]]}

end procedure

Notes
Iterative Weighted Interval Scheduling

Algorithm 22 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time

Require: Prior to first call, p[1..n] has been constructed

\[ M[0] = 0 \]

for \( i \leftarrow 1 \) to \( i \leftarrow n \) do

\[ M[i] = \max\{M[i-1], v_i + M[p[i]]\} \]

end procedure

Notes

- Runtime is \( O(n) \) + time to sort intervals + time to build \( p[] \)
Iterative Weighted Interval Scheduling

Algorithm 23 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time

Require: Prior to first call, p[1..n] has been constructed

\[ M[0] = 0 \]

\[ \text{for } i \leftarrow 1 \text{ to } i \leftarrow n \text{ do} \]

\[ M[i] = \max\{M[i - 1], v_i + M[p[i]]\} \]

end procedure

Notes

• Runtims is \( O(n) + \) time to sort intervals + time to build \( p[\] \)

• Claim: \( p[\] \) can be constructed in \( O(n \log n) \) time
Iterative Weighted Interval Scheduling

**Algorithm 24 Iterated MaxSched**

```plaintext
procedure MAX_SCHED(n)

Require: Intervals have been sorted by increasing finish time
 Require: Prior to first call, p[1..n] has been constructed

M[0] = 0
for i ← 1 to i ← n do
    M[i] = max{M[i - 1], v_i + M[p[i]]}

end procedure
```

Notes

- Runtimes is $O(n) +$ time to sort intervals $+$ time to build $p[]$
- Claim: $p[]$ can be constructed in $O(n \log n)$ time
  (we’ll come back to this)
Iterative Weighted Interval Scheduling

Algorithm 25 Iterated MaxSched

procedure MAXSCHED(n)

Require: Intervals have been sorted by increasing finish time

Require: Prior to first call, p[1..n] has been constructed

\[ M[0] = 0 \]

for \( i \leftarrow 1 \) to \( i \leftarrow n \) do

\[ M[i] = \max\{ M[i - 1], v_i + M[p[i]] \} \]

end procedure

Notes

- Runtims is \( O(n) \) + time to sort intervals + time to build \( p[] \)
- Claim: \( p[] \) can be constructed in \( O(n \log n) \) time (we’ll come back to this)
- Thus time complexity is \( O(n \log n) \); space complexity is \( O(n) \)
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

Method 1: Build partial solutions.
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$. 

Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i - 1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

Method 1: Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i-1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space

Method 2: $S[i]$ just stores a flag indicating whether interval $i$ is part of the optimal solution using only items from $\{1, \ldots, i\}$. 
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i-1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space

**Method 2:** $S[i]$ just stores a flag indicating whether interval $i$ is part of the optimal solution using only items from $\{1, \ldots, i\}$.

Run-time remains $O(n)$ (after the initial sorting).
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $\text{maxSched}(i)$.
- $S[i]$ can be built from $S[i - 1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space

**Method 2:** $S[i]$ just stores a flag indicating whether interval $i$ is part of the optimal solution using only items from $\{1, \ldots, i\}$. Run-time remains $O(n)$ (after the initial sorting).

**Method 3:** Don’t store solution, reconstruct it from $M[]$
**Iterative Weighted Interval Scheduling**

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i - 1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space

**Method 2:** $S[i]$ just stores a flag indicating whether interval $i$ is part of the optimal solution using only items from $\{1, \ldots, i\}$. Run-time remains $O(n)$ (after the initial sorting).

**Method 3:** Don’t store solution, reconstruct it from $M[]$

- If $v_n + M[p[n]] > M[n - 1]$ then include interval $n$ and recursively find rest of solution on intervals $\{1, \ldots, p[n]\}$. 
Iterative Weighted Interval Scheduling

How can we modify to produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in some optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i - 1]$ and $S[p[i]]$

Changes run-time to $O(n^2)$ and adds $O(n^2)$ space.

**Method 2:** $S[i]$ just stores a flag indicating whether interval $i$ is part of the optimal solution using only items from $\{1, \ldots, i\}$. Run-time remains $O(n)$ (after the initial sorting).

**Method 3:** Don’t store solution, reconstruct it from $M[]$

- If $v_n + M[p[n]] > M[n - 1]$ then include interval $n$ and recursively find rest of solution on intervals $\{1, \ldots, p[n]\}$.

Run-time remains $O(n)$ (after the initial sorting).
The Principle of Optimality
The Principle of Optimality

In the Weighted Interval Scheduling Problem we noted that either
The Principle of Optimality

In the Weighted Interval Scheduling Problem we noted that either

- \( maxSched(i) = maxSched(i - 1) \) (item \( i \) not used), or
The Principle of Optimality

In the Weighted Interval Scheduling Problem we noted that either

- \( \text{maxSched}(i) = \text{maxSched}(i - 1) \) (item \( i \) not used), or
- \( \text{maxSched}(i) = v_i + \text{maxSched}(p[i]) \), (item \( i \) was used)
The Principle of Optimality

In the Weighted Interval Scheduling Problem we noted that either

- \( maxSched(i) = maxSched(i - 1) \) (item \( i \) not used), or
- \( maxSched(i) = v_i + maxSched(p[i]) \), (item \( i \) was used)

This is an example of the Principle of Optimality
In the Weighted Interval Scheduling Problem we noted that either

- $\text{maxSched}(i) = \text{maxSched}(i - 1)$ (item $i$ not used), or
- $\text{maxSched}(i) = v_i + \text{maxSched}(p[i])$, (item $i$ was used)

This is an example of the *Principle of Optimality*

An optimal solution to the problem was built from optimal solutions to subproblems.
The Principle of Optimality

In the Weighted Interval Scheduling Problem we noted that either

- \( \text{maxSched}(i) = \text{maxSched}(i - 1) \) (item \( i \) not used), or
- \( \text{maxSched}(i) = v_i + \text{maxSched}(p[i]) \), (item \( i \) was used)

This is an example of the *Principle of Optimality*.

An optimal solution to the problem was built from optimal solutions to subproblems.

This is a common feature of many problems and is a powerful tool in the design of efficient algorithms!
How To Build Predecessor Array

• Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\).

• Sort the \(2n\) pairs by increasing time.

• Now build \(p[]\) by inspecting the pairs in order. We will store the index \(t\) of the largest finish time seen so far.

• Let \(p[1] = 0, t = 0\).

• While there are unscanned items in the list
  • Consider the next item in the list, call it \(x\).
  • If \(x = (s_k, k)\), then set \(p[k] \leftarrow t\).
  • If \(x = (t_k, k)\), then set \(t \leftarrow k\).

This algorithm (clearly) takes \(O(n)\) time in addition to the sorting.
How To Build Predecessor Array

• Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.
- Let \(p[1] = 0, t = 0\).
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order
  
  We will store the index \(t\) of the largest finish time seen so far.
  
  - Let \(p[1] = 0, t = 0\).
  - While there are unscanned items in the list
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.
- Let \(p[1] = 0, t = 0\).
- While there are unscanned items in the list
  - Consider the next item in the list, call it \(x\).
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.

- Let \(p[1] = 0, t = 0\).
- While there are unscanned items in the list
  - Consider the next item in the list, call it \(x\).
  - If \(x = (s_k, k)\), then set \(p[k] \leftarrow t\).
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.

- Let \(p[1] = 0\), \(t = 0\).
- While there are unscanned items in the list
  - Consider the next item in the list, call it \(x\).
  - If \(x = (s_k, k)\), then set \(p[k] \leftarrow t\).
  - If \(x = (t_k, k)\), then set \(t \leftarrow k\).
How To Build Predecessor Array

- Convert each interval \((s_i, t_i)\) into two pairs \((s_i, i)\) and \((t_i, i)\)
- Sort the \(2n\) pairs by increasing time
- Now build \(p[]\) by inspecting the pairs in order

We will store the index \(t\) of the largest finish time seen so far.

- Let \(p[1] = 0\), \(t = 0\).
- While there are unscanned items in the list
  - Consider the next item in the list, call it \(x\).
  - If \(x = (s_k, k)\), then set \(p[k] \leftarrow t\).
  - If \(x = (t_k, k)\), then set \(t \leftarrow k\).

This algorithm (clearly) takes \(O(n)\) time in addition to the sorting.
Outline

Memoizing: A Motivating Example

Weighted Interval Scheduling: A First Dynamic Programming Example

Multi-Parameter Dynamic Programming Problems
  Some Simple Examples: Subset Sum & Knapsack
Subset Sum Decision
Subset Sum Decision

Find a subset of \{27, 25, 14, 22, 12, 8, 23, 16\} that sums to exactly 75.
Subset Sum Decision

Find a subset of \( \{27, 25, 14, 22, 12, 8, 23, 16\} \) that sums to exactly 75.

The Problem: Given numbers \( S = \{s_1, \ldots, s_n\} \) and \( W \), find \( I \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in I} s_i = W \)
Subset Sum Optimization


**Subset Sum Optimization**

Find a subset of \(\{27, 25, 14, 22, 12, 8, 23, 16\}\) with the largest sum possible that doesn’t exceed 65.
**Subset Sum Optimization**

Find a subset of \{27, 25, 14, 22, 12, 8, 23, 16\} with the largest sum possible that doesn’t exceed 65.

The Problem: Given numbers \( S = \{s_1, \ldots, s_n\} \) and \( W \), find \( l \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in l} s_i \) is maximized subject to \( \sum_{i \in l} s_i \leq W \).
Subset Sum Optimization

Find a subset of \{27, 25, 14, 22, 12, 8, 23, 16\} with the largest sum possible that doesn’t exceed 65.

The Problem: Given numbers $S = \{s_1, \ldots, s_n\}$ and $W$, find $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} s_i$ is maximized subject to $\sum_{i \in I} s_i \leq W$

Denote by $S(j, w)$ the largest sum not exceeding $w$ that uses only items from $\{1, \ldots, j\}$


Subset Sum Optimization

Find a subset of \(\{27, 25, 14, 22, 12, 8, 23, 16\}\) with the largest sum possible that doesn’t exceed 65.

The Problem: Given numbers \(S = \{s_1, \ldots, s_n\}\) and \(W\), find \(I \subseteq \{1, \ldots, n\}\) such that \(\sum_{i \in I} s_i\) is maximized subject to \(\sum_{i \in I} s_i \leq W\).

Denote by \(S(j, w)\) the largest sum not exceeding \(w\) that uses only items from \(\{1, \ldots, j\}\)'s

Then

\[
S(j, w) = \max\{S(j - 1, w), w_j + S(j - 1, w - w_j)\}
\]
Subset Sum Optimization

Find a subset of \{27, 25, 14, 22, 12, 8, 23, 16\} with the largest sum possible that doesn’t exceed 65.

The Problem: Given numbers \( S = \{s_1, \ldots, s_n\} \) and \( W \), find \( I \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in I} s_i \) is maximized subject to \( \sum_{i \in I} s_i \leq W \)

Denote by \( S(j, w) \) the largest sum not exceeding \( w \) that uses only items from \( \{1, \ldots, j\} \)

Then

\[
S(j, w) = \max\{S(j - 1, w), w_j + S(j - 1, w - w_j)\}
\]

where

\[
S(j, w) = 0 \text{ if } j = 0 \text{ or } w \leq 0
\]
The Knapsack Problem

The Problem: Given jobs 1, ..., n where each job $i$ has a duration $s_i$ and a value $v_i$, along with a target $W > 0$, find $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} v_i$ is maximized subject to $\sum_{i \in I} s_i \leq W$.
The Knapsack Problem

The Problem: Given jobs 1, . . . , n where each job i has a *duration* \( s_i \) and a *value* \( v_i \), along with a target \( W > 0 \), find \( I \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in I} v_i \) is maximized subject to \( \sum_{i \in I} s_i \leq W \)

Denote by \( S(j, w) \) the largest value using only items from \( \{1, \ldots, j\} \) and having total weight not exceeding \( w \)
The Knapsack Problem

The Problem: Given jobs 1, \ldots, n where each job \( i \) has a duration \( s_i \) and a value \( v_i \), along with a target \( W > 0 \), find \( I \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in I} v_i \) is maximized subject to \( \sum_{i \in I} s_i \leq W \).

Denote by \( S(j, w) \) the largest value using only items from \( \{1, \ldots, j\} \) and having total weight not exceeding \( w \).

Then

\[ S(j, w) = \max\{S(j - 1, w), v_j + S(j - 1, w - w_j)\} \]
The Knapsack Problem

The Problem: Given jobs 1, \ldots, n where each job \( i \) has a \textit{duration} \( s_i \) and a \textit{value} \( v_i \), along with a target \( W > 0 \), find \( I \subseteq \{1, \ldots, n\} \) such that \( \sum_{i \in I} v_i \) is maximized subject to \( \sum_{i \in I} s_i \leq W \).

Denote by \( S(j, w) \) the largest value using only items from \( \{1, \ldots, j\} \) and having total weight not exceeding \( w \).

Then

\[
S(j, w) = \max\{S(j - 1, w), v_j + S(j - 1, w - w_j)\}
\]

where

\[
S(j, w) = 0 \text{ if } j = 0 \text{ or } w \leq 0
\]
Subset Sum vs Knapsack
Subset Sum vs Knapsack

Subset Sum

\[ S(j, w) = \max\{S(j - 1, w), w_j + S(j - 1, w - w_j)\} \]
Subset Sum vs Knapsack

Subset Sum

\[ S(j, w) = \max\{S(j - 1, w), w_j + S(j - 1, w - w_j)\} \]

Knapsack

\[ S(j, w) = \max\{S(j - 1, w), v_j + S(j - 1, w - w_j)\} \]
**Subset Sum vs Knapsack**

**Subset Sum**

\[ S(j, w) = \max \{ S(j - 1, w), w_j + S(j - 1, w - w_j) \} \]

**Knapsack**

\[ S(j, w) = \max \{ S(j - 1, w), v_j + S(j - 1, w - w_j) \} \]

**Pseudo-Polynomial Time**

Both can be computed in time/space \( O(nW) \)