Dynamic Programming I

Algorithm Design & Analysis

Spring 2018
Outline
Announcements

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

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

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

Number of recursive calls made by \( \text{fib}(n) \)
**Fibonacci Numbers**

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

Number of recursive calls made by \( fib(n) \)

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

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

Number of recursive calls made by \( \text{fib}(n) \)

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

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

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

Number of recursive calls made by \( fib(n) \)

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

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

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

Number of recursive calls made by \( fib(n) \)

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

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**

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

Number of recursive calls made by \( \text{fib}(n) \)

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

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

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

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

That is, \( C(n) \) grows exponentially!
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 ≤ 2 then return 1
    \[F_n = F_c = 1\]
    for i ← 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)
    for \( i \leftarrow 3 \) to \( i \leftarrow n \) do
        \( F[i] = F[i - 1] + F[i - 2] \)
    end procedure
Algorithm 4 Fibonacci Table

procedure FIBTABLE\(n\)
    for \(i \leftarrow 3\) to \(i \leftarrow n\) do
        \(F[i] = F[i - 1] + F[i - 2]\)
    end procedure

But what if we don’t know exactly which ones we’ll actually need?
Algorithm 5 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!
**Algorithm 6** 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) $v_i$. 

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

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 maximizes \(\sum_{i \in I} v_i\).
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 maximizes \(\sum_{i \in I} v_i\).

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 maximizes \(\sum_{i \in I} v_i\).

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 maximizes \(\sum_{i \in I} v_i\).

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 maximizes \(\sum_{i \in I} v_i\).

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

• Assume the intervals are sorted by increasing $t$-value.
• For $i \leq n$, let $\text{maxSched}(i)$ be the 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, for each $i > 1$, store the largest $j < i$ such that $t_j \leq s_i$ in a table: $p[i] = j$ (for predecessor).

• 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

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 $\text{maxSched}(i)$ 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 \( \text{maxSched}(i) \) be value of the optimal schedule using only intervals in \( \{1, \ldots, i\} \)
• If \( (s_n, t_n) \) isn't used: \( \text{maxSched}(n) = \text{maxSched}(n - 1) \)
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, for each $i > 1$ store the largest $j < i$ such that $t_j \leq s_i$ in a table: $p[i] = j$ (for predecessor)
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, for each \( i > 1 \) store the largest \( j < i \) such that \( t_j \leq s_i \) in a table: \( p[i] = j \) (for predecessor)
• 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 $maxSched(i)$ 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, for each $i > 1$ store the largest $j < i$ such that $t_j \leq s_i$ in a table: $p[i] = j$ (for predecessor)
- 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: $maxSched(n) = v_n + maxSched(p[n])$
Algorithm 7

MaxSched with Memoizing

procedure maxSched(n)

// Prior to first call, p[1..n] has been constructed
// And a table M[1..n] has been initialized to 0

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
Weighted Interval Scheduling

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

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

Algorithm 9 MaxSched with Memoizing

```plaintext
procedure \text{MAXSCHED}(n)
    // Prior to first call, \( p[1..n] \) has been constructed
    // And a table \( M[1..n] \) has been initialized to 0
    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
```
Algorithm 10 Iterated MaxSched

procedure MAXSCHED(n)
    // 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
Iterative Weighted Interval Scheduling

Algorithm 11 Iterated MaxSched

procedure MAX_SCHED(n)
   // 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

• If the intervals are sorted, p[] can be built in O(n) time (convince yourselves!)
• Thus the algorithm takes O(n) space and O(n log n) time.
• In fact, assuming we've sorted the (endpoints) of the intervals, it takes O(n) time!
Algorithm 12 Iterated MaxSched

procedure MAXSCHED(n)
    // 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

• If the intervals are sorted, $p[]$ can be built in $O(n)$ time (convince yourselves!)
Iterative Weighted Interval Scheduling

Algorithm 13 Iterated MaxSched

procedure MAXSCHED(n)
    // 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

• If the intervals are sorted, p[] can be built in $O(n)$ time (convince yourselves!)
• Thus the algorithm takes $O(n)$ space and $O(n \log n)$ time.
Iterative Weighted Interval Scheduling

Algorithm 14 Iterated MaxSched

procedure MAXSCHED(n)
    // 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

- If the intervals are sorted, p[] can be built in $O(n)$ time (convince yourselves!)
- Thus the algorithm takes $O(n)$ space and $O(n \log n)$ time.
- In fact, assuming we’ve sorted the (endpoints) of the intervals, it takes $O(n)$ timed!
Iterative Weighted Interval Scheduling

How could we modify the method to also produce the optimal set of intervals?
Iterative Weighted Interval Scheduling

How could we modify the method to also produce the optimal set of intervals?

**Method 1:** Build partial solutions.
How could we modify the method to also produce the optimal set of intervals?

**Method 1:** Build partial solutions.

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

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

How could we modify the method to also produce the optimal set of intervals?

**Method 1:** Build partial solutions.

- Compute table $S[]$, where $S[i]$ holds intervals in optimal solution to $maxSched(i)$.
- $S[i]$ can be built from $S[i - 1]$ and $S[p[i]]$
How could we modify the method to also produce the optimal set of intervals?

**Method 1:** Build partial solutions.

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

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

How could we modify the method to also produce the optimal set of intervals?

Method 1: Build partial solutions.

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

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

Method 2: Reconstruct from $M[]$
How could we modify the method to also produce the optimal set of intervals?

**Method 1:** Build partial solutions.

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

Changes run-time to \( O(n^2) \)

**Method 2:** Reconstruct from \( M[] \)

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

How could we modify the method to also produce the optimal set of intervals?

Method 1: Build partial solutions.

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

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

Method 2: Reconstruct from $M[]$

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

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

In the Weighted Interval Scheduling Problem we noted that either

\[ \text{maxSched}(i) = \text{maxSched}(i-1) \quad \text{(item } i \text{ not used)} \]

or

\[ \text{maxSched}(i) = v_i + \text{maxSched}(p[i]) \quad \text{(item } i \text{ was used)} \]

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!
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

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