

Algorithms Week 6

Ljubomir Perković, DePaul University

Dynamic Programming

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- 2 Build solutions to your recurrence from the bottom up

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems
 - b Choose a memoization data structure

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems
 - b Choose a memoization data structure
 - c Identify dependencies

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems
 - b Choose a memoization data structure
 - c Identify dependencies
 - d Find a good evaluation order

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems
 - b Choose a memoization data structure
 - c Identify dependencies
 - d Find a good evaluation order
 - e Write down the algorithm

The basic idea behind dynamic programming is **recursion without repetition**.

To develop a dynamic algorithm:

- ① Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm
- ② Build solutions to your recurrence from the bottom up
 - a Identify the subproblems
 - b Choose a memoization data structure
 - c Identify dependencies
 - d Find a good evaluation order
 - e Write down the algorithm
 - f Analyze space and running time

Subset sum is a classic problem:

Input: An array $X[1..n]$ of positive integers and an integer T .

Output: A subset of X that sums to T .

Subset sum is a classic problem:

Input: An array $X[1..n]$ of positive integers and an integer T .

Output: A subset of X that sums to T .

Example: Let $X = [4, 7, 6, 3, 1]$ and $T = 10$.

Subset sum is a classic problem:

Input: An array $X[1..n]$ of positive integers and an integer T .

Output: A subset of X that sums to T .

Example: Let $X = [4, 7, 6, 3, 1]$ and $T = 10$.

- $4 + 6 = 10$, so $\{4, 6\}$ is a valid solution.

Subset sum is a classic problem:

Input: An array $X[1..n]$ of positive integers and an integer T .

Output: A subset of X that sums to T .

Example: Let $X = [4, 7, 6, 3, 1]$ and $T = 10$.

- $4 + 6 = 10$, so $\{4, 6\}$ is a valid solution.
- $6 + 3 + 1 = 10$, so $\{6, 3, 1\}$ is a valid solution.

Subset sum is a classic problem:

Input: An array $X[1..n]$ of positive integers and an integer T .

Output: A subset of X that sums to T .

Example: Let $X = [4, 7, 6, 3, 1]$ and $T = 10$.

- $4 + 6 = 10$, so $\{4, 6\}$ is a valid solution.
- $6 + 3 + 1 = 10$, so $\{6, 3, 1\}$ is a valid solution.
- $4 + 3 + 1 = 8$, so $\{4, 3, 1\}$ is **not** a valid solution.

Dynamic programming algorithm development step 1

To develop a dynamic algorithm, first formulate the problem recursively

- a Describe the problem that you want to solve recursively
- b Give a clear recursive formula or algorithm

Dynamic programming algorithm development step 1

To develop a dynamic algorithm, first formulate the problem recursively

- a Describe the problem that you want to solve recursively
- b Give a clear recursive formula or algorithm

Define $SS(i, t)$ to be *True* if some subset of $X[i..n]$ sums to t .

Dynamic programming algorithm development step 1

To develop a dynamic algorithm, first formulate the problem recursively

- a Describe the problem that you want to solve recursively
- b Give a clear recursive formula or algorithm

Define $SS(i, t)$ to be *True* if some subset of $X[i..n]$ sums to t .
Then

$$SS(i, t) = \begin{cases} \textit{True}, & \text{if } t = 0 \\ \textit{False}, & \text{if } i > n \\ SS(i + 1, t), & \text{if } t < X[i] \\ SS(i + 1, t) \vee SS(i + 1, t - X[i]), & \text{otherwise} \end{cases}$$

Dynamic programming algorithm development step 1

To develop a dynamic algorithm, first formulate the problem recursively

- a Describe the problem that you want to solve recursively
- b Give a clear recursive formula or algorithm

Define $SS(i, t)$ to be *True* if some subset of $X[i..n]$ sums to t .
Then

$$SS(i, t) = \begin{cases} \textit{True}, & \text{if } t = 0 \\ \textit{False}, & \text{if } i > n \\ SS(i + 1, t), & \text{if } t < X[i] \\ SS(i + 1, t) \vee SS(i + 1, t - X[i]), & \text{otherwise} \end{cases}$$

We want to compute $SS(1, T)$.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are of type $SS(i, t)$ where $1 \leq i \leq n + 1$ and $0 \leq t \leq T \dots$

Dynamic programming algorithm development step 2

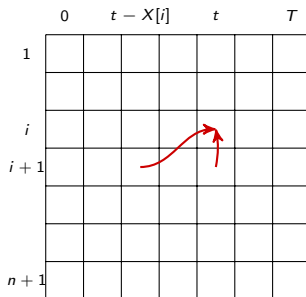
Build solutions to your recurrence from the bottom up

- a Recursive subproblems are of type $SS(i, t)$ where $1 \leq i \leq n + 1$ and $0 \leq t \leq T$...
- b ... so we need a two-dimensional array $SS[1..n+1, 0..T]$.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

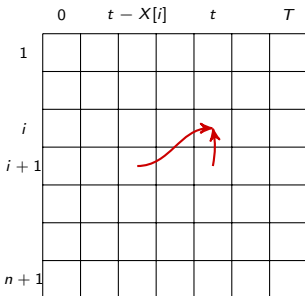
- a Recursive subproblems are of type $SS(i, t)$ where $1 \leq i \leq n + 1$ and $0 \leq t \leq T$...
- b ... so we need a two-dimensional array $SS[1..n+1, 0..T]$.
- c Each entry $SS[i, t]$ depends only on entries $SS(i+1, t)$ and $SS(i+1, t-X[i])$...



Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are of type $SS(i, t)$ where $1 \leq i \leq n + 1$ and $0 \leq t \leq T$...
- b ... so we need a two-dimensional array $SS[1..n+1, 0..T]$.
- c Each entry $SS[i, t]$ depends only on entries $SS(i+1, t)$ and $SS(i+1, t-X[i])$...

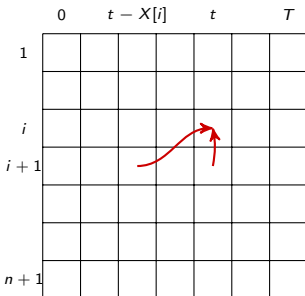


- d ... so we need to fill the 2D-array row-by-row bottom-up.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are of type $SS(i, t)$ where $1 \leq i \leq n + 1$ and $0 \leq t \leq T$...
- b ... so we need a two-dimensional array $SS[1..n+1, 0..T]$.
- c Each entry $SS[i, t]$ depends only on entries $SS(i+1, t)$ and $SS(i+1, t-X[i])$...



- d ... so we need to fill the 2D-array row-by-row bottom-up.

FastSubsetSum (via dynamic programming)

```
FastSubsetSum(X[1..n], T):  
  S[n+1, 0] ← True  
  for t ← 1 to T  
    S[n+1, t] ← False  
  for i ← n downto 1  
    S[i, 0] ← True  
    for t ← 1 to X[i] - 1  
      S[i, t] ← S[i+1, t]  
    for t ← X[i] to T  
      S[i, t] ← S[i+1, t] ∨ S[i+1, t-X[i]]  
  
return S[1, T]
```

FastSubsetSum (via dynamic programming)

```
FastSubsetSum(X[1..n], T):  
  S[n+1, 0] ← True  
  for t ← 1 to T  
    S[n+1, t] ← False  
  for i ← n downto 1  
    S[i, 0] ← True  
    for t ← 1 to X[i] - 1  
      S[i, t] ← S[i+1, t]  
    for t ← X[i] to T  
      S[i, t] ← S[i+1, t] ∨ S[i+1, t-X[i]]  
  
return S[1, T]
```

Running time?

FastSubsetSum (via dynamic programming)

```
FastSubsetSum(X[1..n], T):  
  S[n+1, 0] ← True  
  for t ← 1 to T  
    S[n+1, t] ← False  
  for i ← n downto 1  
    S[i, 0] ← True  
    for t ← 1 to X[i] - 1  
      S[i, t] ← S[i+1, t]  
    for t ← X[i] to T  
      S[i, t] ← S[i+1, t] ∨ S[i+1, t-X[i]]  
  
return S[1, T]
```

Running time? $O(nT)$

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

FOOD

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

MOOD

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

MOOD

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

MOND

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

MONED

The edit distance between two strings is the minimum number of

- letter insertions,
- letter deletions, and
- letter substitutions

required to transform one string into the other.

For example, the edit distance between **FOOD** and **MONEY** is 4:

MONEY

Formal problem specification:

Input: Two strings $A[1..m]$ and $B[1..n]$

Output: The edit distance between A and B

The edit distance between **ALGORITHM** and **ALTRUISTIC**,
using another kind of visualization:

The edit distance between **ALGORITHM** and **ALTRUISTIC**,
using another kind of visualization:

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

The edit distance between **ALGORITHM** and **ALTRUISTIC**, using another kind of visualization:

A	L	G	O	R	I		T	H	M	
A	L		T	R	U	I	S	T	I	C

Columns with

- a gap in the top word represent the insertion of a letter

The edit distance between **ALGORITHM** and **ALTRUISTIC**, using another kind of visualization:

A	L	G	O	R	I		T	H	M	
A	L		T	R	U	I	S	T	I	C

Columns with

- a gap in the top word represent the insertion of a letter
- a gap in the bottom word represents the deletion of a letter

The edit distance between **ALGORITHM** and **ALTRUISTIC**, using another kind of visualization:

A	L	G	O	R	I		T	H	M	
A	L		T	R	U	I	S	T	I	C

Columns with

- a gap in the top word represent the insertion of a letter
- a gap in the bottom word represents the deletion of a letter
- with two different characters correspond to substitutions

To develop a dynamic algorithm:

- 1 Formulate the problem recursively

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

If we remove the last column,

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

A	L	G	O	R		I		T	H
A	L		T	R	U	I	S	T	I

If we remove the last column,

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

A	L	G	O	R		I		T	H
A	L		T	R	U	I	S	T	I

If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

A	L	G	O	R		I		T	H
A	L		T	R	U	I	S	T	I

If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.

In other words, once we decide what should happen in the last column, the **recursion fairy** will figure out the rest.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively

A	L	G	O	R		I		T	H
A	L		T	R	U	I	S	T	I

If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.

In other words, once we decide what should happen in the last column, the **recursion fairy** will figure out the rest.

Let $Edit(i, j)$ be the edit distance between $A[1..i]$ and $B[1..j]$.

So what can happen in the last column in general?

So what can happen in the last column in general?

ALGOR
ALTRU

So what can happen in the last column in general?

ALGOR			
ALTR		U	

An insertion, i.e. the last entry in top row is empty.

So what can happen in the last column in general?

ALGOR			
ALTR		U	

An insertion, i.e. the last entry in top row is empty. Then

$$Edit(i, j) = 1 + Edit(i, j - 1)$$

So what can happen in the last column in general?

So what can happen in the last column in general?

ALGOR
ALTRU

So what can happen in the last column in general?

ALGO		R	
ALTRU			

A deletion, i.e. the last entry in bottom row is empty.

So what can happen in the last column in general?

ALGO		R	
ALTRU			

A deletion, i.e. the last entry in bottom row is empty. Then

$$Edit(i, j) = 1 + Edit(i - 1, j)$$

So what can happen in the last column in general?

So what can happen in the last column in general?

ALGOR
ALTRU

So what can happen in the last column in general?

ALGO		R	
ALTR		U	

A substitution, i.e. the last entry in both rows is non-empty.

So what can happen in the last column in general?

ALGO		R	
ALTR		U	

A substitution, i.e. the last entry in both rows is non-empty. Then

$$Edit(i, j) = 1 + Edit(i - 1, j - 1)$$

if the characters are different.

So what can happen in the last column in general?

So what can happen in the last column in general?

ALGOR
ALTR

So what can happen in the last column in general?

ALGO		R	
ALT		R	

A substitution, i.e. the last entry in both rows is non-empty.

So what can happen in the last column in general?

ALGO		R	
ALT		R	

A substitution, i.e. the last entry in both rows is non-empty. Then

$$Edit(i, j) = Edit(i - 1, j - 1)$$

if the characters are the same. The substitution is free.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \begin{cases} i, & \text{if } j = 0 \end{cases}$$

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \end{cases}$$

Edit function recurrence

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i, j - 1) + 1 \end{array} \right. \end{cases}$$

Edit function recurrence

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j - 1) + 1 \\ Edit(i - 1, j) + 1 \end{cases} & \text{otherwise} \end{cases}$$

Edit function recurrence

To develop a dynamic algorithm:

- 1 Formulate the problem recursively
 - a Describe the problem that you want to solve recursively
 - b Give a clear recursive formula or algorithm

$Edit(i, j)$ is the edit distance between $A[1..i]$ and $B[1..j]$.

$$Edit(i, j) = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j - 1) + 1 \\ Edit(i - 1, j) + 1 \\ Edit(i - 1, j - 1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are $\text{Edit}(i, j)$ with $0 \leq i \leq m$ and $0 \leq j \leq n$...

Dynamic programming algorithm development step 2

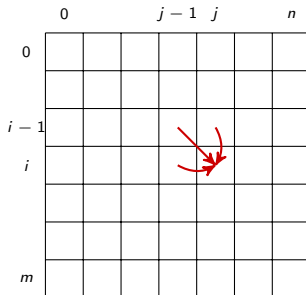
Build solutions to your recurrence from the bottom up

- a Recursive subproblems are $\text{Edit}(i, j)$ with $0 \leq i \leq m$ and $0 \leq j \leq n$...
- b ... so we need a two-dimensional array $\text{Edit}[0..m, 0..n]$.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

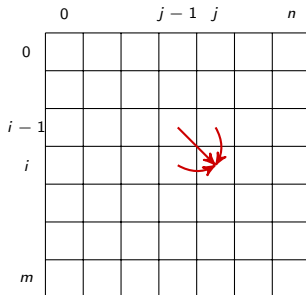
- a Recursive subproblems are $\text{Edit}(i, j)$ with $0 \leq i \leq m$ and $0 \leq j \leq n$...
- b ... so we need a two-dimensional array $\text{Edit}[0..m, 0..n]$.
- c Each entry $\text{Edit}[i, j]$ depends only on entries $\text{Edit}[i-1, j]$, $\text{Edit}[i, j-1]$, and $\text{Edit}[i-1, j-1]$...



Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are $\text{Edit}(i, j)$ with $0 \leq i \leq m$ and $0 \leq j \leq n$...
- b ... so we need a two-dimensional array $\text{Edit}[0..m, 0..n]$.
- c Each entry $\text{Edit}[i, j]$ depends only on entries $\text{Edit}[i-1, j]$, $\text{Edit}[i, j-1]$, and $\text{Edit}[i-1, j-1]$...

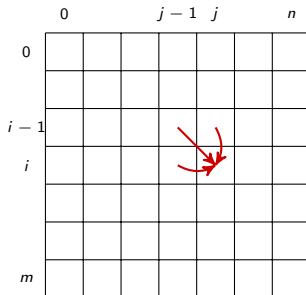


- d ... so we need to fill the two-dimensional array top-down left-to-right.

Dynamic programming algorithm development step 2

Build solutions to your recurrence from the bottom up

- a Recursive subproblems are $\text{Edit}(i, j)$ with $0 \leq i \leq m$ and $0 \leq j \leq n$...
- b ... so we need a two-dimensional array $\text{Edit}[0..m, 0..n]$.
- c Each entry $\text{Edit}[i, j]$ depends only on entries $\text{Edit}[i-1, j]$, $\text{Edit}[i, j-1]$, and $\text{Edit}[i-1, j-1]$...



- d ... so we need to fill the two-dimensional array top-down left-to-right.

Edit distance dynamic programming algorithm

```
EditDistance(A[1 .. m], B[1 .. n]):  
  for j ← 0 to n  
    Edit[0, j] ← j  
  for i ← 1 to m  
    Edit[i, 0] ← i  
    for j ← 1 to n  
      ins ← Edit[i, j-1] + 1  
      del ← Edit[i-1, j] + 1  
      if A[i] = B[j]  
        rep ← Edit[i-1, j-1]  
      else  
        rep ← Edit[i-1, j-1] + 1  
      Edit[i, j] ← min{ins, del, rep}  
  return Edit[m, n]
```

Edit distance dynamic programming algorithm

```
EditDistance(A[1 .. m], B[1 .. n]):  
  for j ← 0 to n  
    Edit[0, j] ← j  
  for i ← 1 to m  
    Edit[i, 0] ← i  
    for j ← 1 to n  
      ins ← Edit[i, j-1] + 1  
      del ← Edit[i-1, j] + 1  
      if A[i] = B[j]  
        rep ← Edit[i-1, j-1]  
      else  
        rep ← Edit[i-1, j-1] + 1  
      Edit[i, j] ← min{ins, del, rep}  
  return Edit[m, n]
```

Running time?

Edit distance dynamic programming algorithm

```
EditDistance(A[1 .. m], B[1 .. n]):  
  for j ← 0 to n  
    Edit[0, j] ← j  
  for i ← 1 to m  
    Edit[i, 0] ← i  
    for j ← 1 to n  
      ins ← Edit[i, j-1] + 1  
      del ← Edit[i-1, j] + 1  
      if A[i] = B[j]  
        rep ← Edit[i-1, j-1]  
      else  
        rep ← Edit[i-1, j-1] + 1  
      Edit[i, j] ← min{ins, del, rep}  
  return Edit[m, n]
```

Running time? $O(mn)$

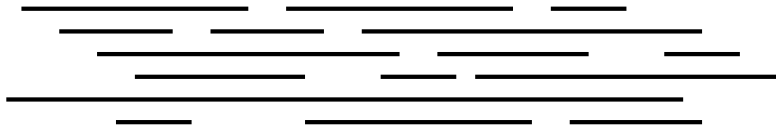
Activity Selection Problem

Suppose activities $1, 2, \dots, n$ are scheduled at different times of a single day.

Activity Selection Problem

Suppose activities $1, 2, \dots, n$ are scheduled at different times of a single day.

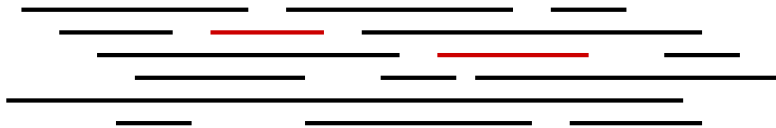
Each activity i has start time s_i and finish time f_i such that $s_i < f_i$; activity i can be represented as (s_i, f_i) .



Activity Selection Problem

Suppose activities $1, 2, \dots, n$ are scheduled at different times of a single day.

Each activity i has start time s_i and finish time f_i such that $s_i < f_i$; activity i can be represented as (s_i, f_i) .

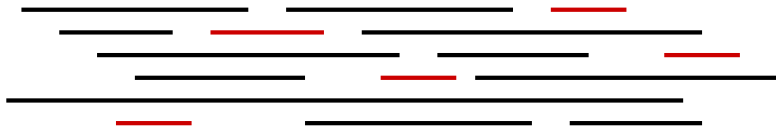


Two activities (s_i, f_i) and (s_j, f_j) are compatible if they do not overlap, i.e. if either $f_j < s_i$ or $f_i < s_j$.

Activity Selection Problem

Suppose activities $1, 2, \dots, n$ are scheduled at different times of a single day.

Each activity i has start time s_i and finish time f_i such that $s_i < f_i$; activity i can be represented as (s_i, f_i) .



Two activities (s_i, f_i) and (s_j, f_j) are compatible if they do not overlap, i.e. if either $f_j < s_i$ or $f_i < s_j$.

Goal: choose the largest possible set of compatible activities.

Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

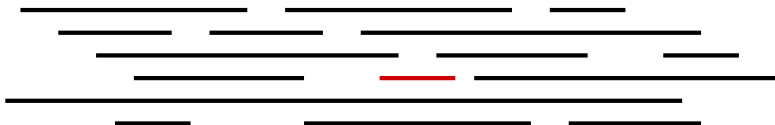
Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

Solution via recursive backtracking:

If activity i is in an optimal solution



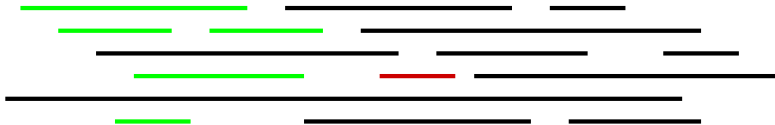
Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

Solution via recursive backtracking:

If activity i is in an optimal solution then so is the optimal solution for the set of activities that end before s_i .



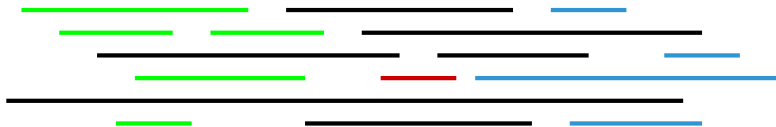
Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

Solution via recursive backtracking:

If activity i is in an optimal solution then so is the optimal solution for the set of activities that end before s_i and the optimal solution the set of activities that start after f_i .



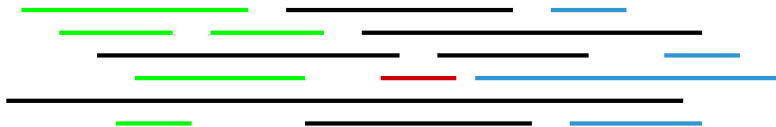
Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

Solution via recursive backtracking:

If activity i is in an optimal solution then so is the optimal solution for the set of activities that end before s_i and the optimal solution the set of activities that start after f_i .



Running time: $O(2^n)$

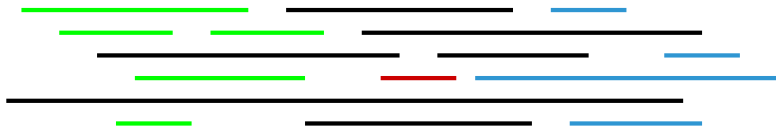
Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

Output: The largest possible set of compatible activities.

Solution via dynamic programming:

If activity i is in an optimal solution then so is the optimal solution for the set of activities that end before s_i and the optimal solution the set of activities that start after f_i .



Running time: $O(n^3)$

Activity Selection Problem

Input: List of activities $1, 2, 3, \dots, n$ scheduled at time intervals $(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)$.

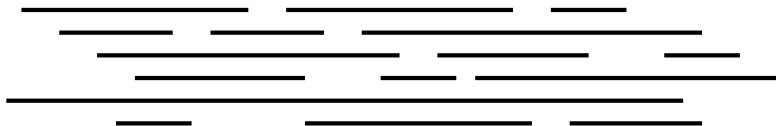
Output: The largest possible set of compatible activities.

Solution via the greedy method:

Running time: $O(n \log n)$

The greedy method

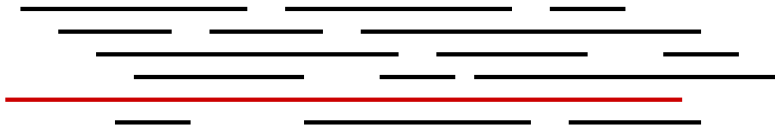
We try “greedy” approaches:



The greedy method

We try “greedy” approaches:

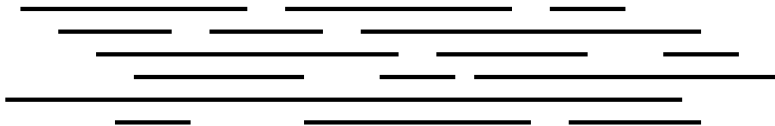
- Choose **activity** that starts earliest...



The greedy method

We try “greedy” approaches:

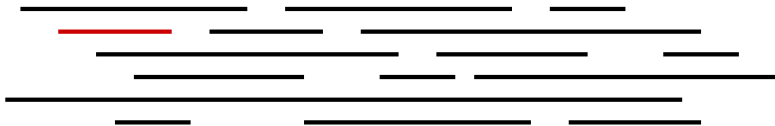
- Choose activity that starts earliest...



The greedy method

We try “greedy” approaches:

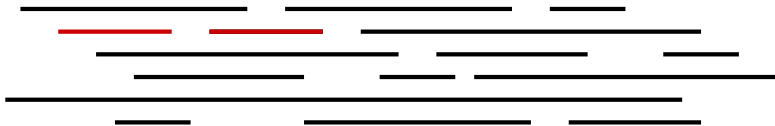
- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



The greedy method

We try “greedy” approaches:

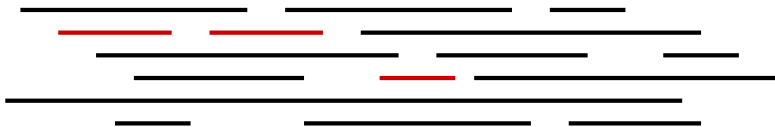
- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



The greedy method

We try “greedy” approaches:

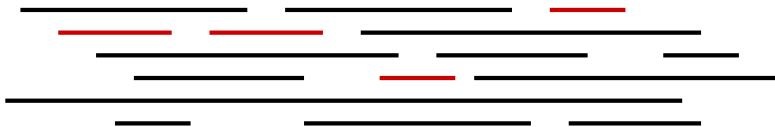
- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



The greedy method

We try “greedy” approaches:

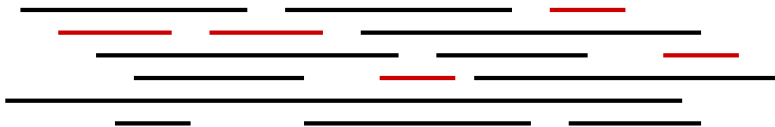
- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



The greedy method

We try “greedy” approaches:

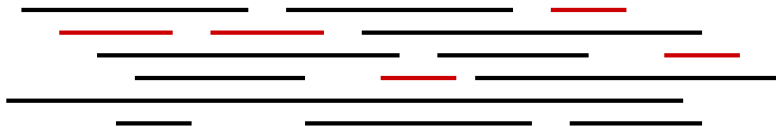
- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



The greedy method

We try “greedy” approaches:

- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...

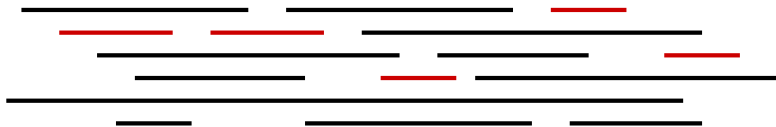


Seems to be working...

The greedy method

We try “greedy” approaches:

- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...

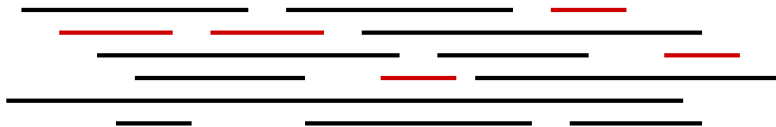


Seems to be working... but we have to prove this.

The greedy method

We try “greedy” approaches:

- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



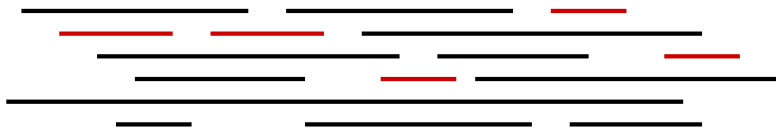
Seems to be working... but we have to prove this. **Assuming activities are sorted by finishing time**, we need to show that:

- 1 There exists a largest set A of compatible activities containing activity 1.

The greedy method

We try “greedy” approaches:

- ~~Choose activity that starts earliest...~~
- Choose activity that **finishes** earliest...



Seems to be working... but we have to prove this. **Assuming activities are sorted by finishing time**, we need to show that:

- ① There exists a largest set A of compatible activities containing activity 1.
- ② If A is a largest set of activities containing 1 then $A - \{1\}$ is a largest set of compatible activities for $\{(s_i, f_i) : s_i \geq f_1\}$.

- 1 There exists a largest set A of compatible activities containing activity 1.

- 1 There exists a largest set A of compatible activities containing activity 1.
- 2 If A is a largest set of activities containing 1 then $A - \{1\}$ is a largest set of compatible activities for $\{(s_i, f_i) : s_i \geq f_1\}$.

Let s be an array of activity starting times and let f be an array of activity finishing times, **both sorted by finishing time**.

In other words, $f[1] \leq f[2] \leq \dots \leq f[n]$.

Let s be an array of activity starting times and let f be an array of activity finishing times, **both sorted by finishing time**.

In other words, $f[1] \leq f[2] \leq \dots \leq f[n]$.

```
GreedyASP(s, f, n)
  A ← {1}
  j ← 1
  for i ← 2 to n
    if s[i] ≥ f[j] then
      A ← A U {i}
      j ← i
  return A
```

Let s be an array of activity starting times and let f be an array of activity finishing times, **both sorted by finishing time**.

In other words, $f[1] \leq f[2] \leq \dots \leq f[n]$.

```
GreedyASP(s, f, n)
  A ← {1}
  j ← 1
  for i ← 2 to n
    if s[i] ≥ f[j] then
      A ← A U {i}
      j ← i
  return A
```

Running time?

Let s be an array of activity starting times and let f be an array of activity finishing times, **both sorted by finishing time**.

In other words, $f[1] \leq f[2] \leq \dots \leq f[n]$.

```
GreedyASP(s, f, n)
  A ← {1}
  j ← 1
  for i ← 2 to n
    if s[i] ≥ f[j] then
      A ← A U {i}
      j ← i
  return A
```

Running time? $\Theta(n)$, but only if not including the $\Theta(n \log n)$ time required to sort s and f .

A problem can be solved using a greedy algorithm if it satisfies these conditions:

Greedy choice property: the optimal solution can be constructed from locally optimal choices. This must always be proved before you construct a greedy algorithm for the problem.

Optimal substructure: The optimal solutions contain the optimal solutions to its subproblems.