Dynamic Programming

- Input : Array A[1,…….,n]

- Output : A subset of A, A′ such that if A[i] is in A′, then A[i+1] is not in A′ and sum of elements in A′ is maximised.

Step 1 : Define a Subproblem such that solving all the subproblems will help us solve the actual problem.

Step 1 : Define a Subproblem such that solving all the sub-problems will help us solve the actual problem.

OPT[j]

- Input : Array A[1,…….,j]
- Output : A subset of A, A′ such that if A[i] is in A′, then A[i+1] is not in A′ and sum of elements in A′ is maximised.

Step 1 : Define a Subproblem such that solving all the sub-problems will help us solve the actual problem.

OPT[j]

- Input : Array A[1,…….,j]
- Output : A subset of A, A′ such that if A[i] is in A′, then A[i+1] is not in A′ and sum of elements in A′ is maximised.
- OPT[n] - desired solution

Step 1 : Define a Subproblem such that solving all the sub-problems will help us solve the actual problem.

- There are only a “small” number of sub-problems
- There is a natural ordering on sub problems from “smallest” to “largest”

Step 2 : Write a recurrence for the subproblem(s).

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a “guess” about the solution and solve a smaller sub-problem for each guess.

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a “guess” about the solution and solve a smaller sub-problem for each guess.

1. A[j] is in solution

2. A[j] is not in solution

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a “guess” about the solution and solve a smaller sub-problem for each guess.

1. A[j] is in solution

2. A[j] is not in solution

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a "guess" about the solution and solve a smaller sub-problem for each guess.

1. A[j] is in solution : OPT[j] = A[j]+OPT[j-2]

2. A[j] is not in solution : OPT[j] = OPT[j-1]

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a "guess" about the solution and solve a smaller sub-problem for each guess.

1. A[j] is in solution : OPT[j] = A[j]+OPT[j-2]

2. A[j] is not in solution : OPT[j] = OPT[j-1]

OPT[j] = max { A[j]+OPT[j-2], OPT[j-1] }

Step 2 : Write a recurrence for the subproblem(s).

For this, we usually have to make a "guess" about the solution and solve a smaller sub-problem for each guess.

1. A[j] is in solution : $OPT[j] = A[j]+OPT[j-2]$
2. A[j] is not in solution : $OPT[j] = OPT[j-1]$

$OPT[j] = \max \{ A[j]+OPT[j-2], OPT[j-1] \}$

$OPT[1] = A[1]$

$OPT[2] = \max\{A[1], A[2]\}$

Step 3 : Prove the correctness of the recurrence (usually by induction)

Step 3 : Prove the correctness of the recurrence (usually by induction)

OPT[j] = max { A[j]+OPT[j-2], OPT[j-1] }

By induction on j :

Step 4 : Solve the sub-problems in a bottom-up fashion

Step 4 : Solve the sub-problems in a bottom-up fashion

$$OPT[j] = \max \{ A[j]+OPT[j-2], OPT[j-1] \}$$

Step 4 : Solve the sub-problems in a bottom-up fashion

```
OPT[1] = A[1]

OPT[2] = max{A[1], A[2]}

for j = 3 to n

  OPT[j] = max { A[j]+OPT[j-2], OPT[j-1] }

return OPT[n]
```

Step 5 : Constructing the actual solution

```
OPT[1] = A[1]

OPT[2] = max{A[1], A[2]}

for j = 3 to n

   OPT[j] = max { A[j]+OPT[j-2], OPT[j-1] }

   if OPT[j] = A[j]+OPT[j-2], then flag[j] =1

   else flag[j] = 0

s=n

while s > 0

   if flag [s] = 1,

      Add A[s] to the solution; s = s-2

   else

      s=s-1
```

Running Time :

Time taken to compute a sub-problem (Assuming solutions of smaller sub-problems are known)

X

number of sub-problems

Weighted Interval Scheduling

Input : Set of n intervals, I

$$w : I \to \mathbb{R}$$

Output : $I' \subseteq I$ such that $\forall i_1, i_2 \in I', i_1 \cap i_2 = \emptyset$ and $\sum_{i \in I'} w(i)$ is maximised.

Weighted Interval Scheduling

Input : Set of n intervals, I

$$w : I \to \mathbb{R}$$

Output : $I' \subseteq I$ such that $\forall i_1, i_2 \in I', i_1 \cap i_2 = \emptyset$ and $\sum_{i \in I'} w(i)$ is maximised.

Order the intervals by their finishing point

$I = \{i_1, i_2, \ldots, i_n\}$

OPT [j] - solution for the input $I_j = \{i_1, i_2, \ldots, i_j\}$

Weighted Interval Scheduling

$$OPT[j] = \max OPT[j-1], w(i_j) + OPT[p_j]$$

p_j = index of the last interval that does not intersect with i_j

Weighted Interval Scheduling

$$OPT[j] = \max OPT[j-1], w(i_j) + OPT[p_j]$$

p_j = index of the last interval that does not intersect with i_j

Weighted Interval Scheduling

$$OPT[j] = \max OPT[j-1], w(i_j) + OPT[p_j]$$

p_j = index of the last interval that intersects with i_j

Base case?

Weighted Interval Scheduling

$$OPT[j] = \max OPT[j-1], w(i_j) + OPT[p_j]$$

p_j = index of the last interval that intersects with i_j

Proof of correctness by induction on j :

```
OPT[0] = 0

OPT[1] = w(i_1)

compute p_j for all j

for j = 2 to n

    OPT[j] = max OPT[j − 1], w(i_j) + OPT[p_j]

    if OPT[j] = w(i_j) + OPT[p_j], then flag[j] =1

    else flag[j] = 0

s=n

while s > 0

    if flag [s] = 1,

        Add i_s to the solution; s = p_s

    else

        s=s-1
```

Running Time ?

Longest Increasing Subsequence

Input : An array of n integers, A

Output : The length of the longest increasing subsequence in A

Longest Increasing Subsequence

Input : An array of n integers, A

Output : The length of the longest increasing subsequence in A

subsequence - An ordered array B such that all the elements of B are in A, for all i, B[i] appears before B[i+1] in A.

increasing subsequence : A subsequence B such that B[i] < B[i+1], for all i.

Longest Increasing Subsequence

OPT[i] : optimum solution for A[1...i], that contains A[i]

Longest Increasing Subsequence

OPT[i] : optimum solution for A[1...i], that contains A[i]

$\max_{1 \le i \le n} OPT[i]$ gives the final solution

Longest Increasing Subsequence

OPT[i] = $\max_{j<i, A[j]<A[i]} 1 + OPT[j]$

Longest Increasing Subsequence

Correctness of recurrence:

Longest Increasing Subsequence

Bottom-up implementation :

Longest Increasing Subsequence

Constructing the actual solution :

Longest Increasing Subsequence

Running Time :