Algorithm Design & Analysis

Natthapong Jungteerapanich

Dynamic Programming

Revisiting Divide and Conquer

- Divide and conquer is a powerful algorithm design technique. It involves breaking down a problem into subproblems, solve the subproblems, and then combine the solutions to the subproblems to obtain the solution to the original problem.
- This process can be naturally implemented using recursion. But for many problems, a simple non-recursive implementation is possible. For example, binary search:

```
def binsearch_rec(A,lo,hi,x):
  if hi<lo:
    return None
  else:
    mid = (hi+lo)//2
    if x == A[mid]:
      return mid
    elif x < A[mid]:
      return binsearch_rec(A,lo,mid-1,x)
    else:
      return binsearch_rec(A,mid+1,hi,x)
```

```
def binsearch(A,lo,hi,x):
  while lo<=hi:
    mid = (hi+lo)//2
    if x == A[mid]:
      return mid
    elif x < A[mid]:
      hi = mid-1
    else:
      lo = mid+1
  return None
```

Revisiting Divide and Conquer

- For some problems, mainly those involving breaking down and solving two or more subproblems, a non-recursive implementation is possible but more complicated.

```
function quick_sort(A,lo,hi)
  if lo>=hi then
    return
  else
    pivot = select_pivot(A,lo,hi)
    mid1,mid2 = partition(A,lo,hi,pivot)
    quick_sort(A,lo,mid1-1)
    quick_sort(A,mid2+1,hi)
```

```
function merge_sort(P[0…N-1])
    if N <= 1 then
        return P
    else
        M = ⌊N/2⌋
        SL = merge_sort(P[0…M-1])
        SR = merge_sort(P[M…N])
        return merge(SL,SR)
```

Overlapping Subproblems

- Notice that in each D&C algorithm that we studied so far, a problem is divided into subproblems that are non-overlapping, i.e. completely separate.
- Consider the following problem.
- **Problem**: How many bit strings of length N that do <u>not</u> have two consecutive 0s are there?
- Let us define a bit string of length n ($n \geq 0$) to be "good" if and only if it does not contain any two consecutive 0s.
- Let G_n be the set of all good bit strings of length n.
- Since all bit strings of length 0 or 1 are good, we have

$$|G_0| = |\{\epsilon\}| = 1 \qquad \text{and} \qquad |G_1| = |\{0,1\}| = 2.$$

Overlapping Subproblems

- Assume that $n \geq 2$. Let us partition G_n into two classes: GA_n and GB_n
  - GA_n contains good bit strings ending with 0 and
  - GB_n contains good bit strings ending with 1:

GA_n = Good bit strings ending with 0	**GB_n = Good bit strings ending with 1**
• Example: 10110, 01010, 11110, 11010	• Example: 10101, 10111, 01011, 01101

Overlapping Subproblems

GA_n = **Good bit strings ending with 0** $(n \geq 2)$	GB_n = **Good bit strings ending with 1** $(n \geq 2)$
• Example: 10110, 01010, 11110, 11010 • This set GA_n must contain all good bit strings of the form $\boxed{\quad u \quad}\boxed{10}$ where u is a good bit string of length $n-2$. • Hence, $\lvert GA_n \rvert = \lvert G_{n-2} \rvert$.	• Example: 10101, 10111, 01011, 01101 • This set G_n must contain all good bit strings of the form $\boxed{\quad u \quad}\boxed{1}$ where u is a good bit string of length $n-1$. Hence, $\lvert GB_n \rvert = \lvert G_{n-1} \rvert$.

- Thus, in total, $|G_n| = |GA_n| + |GB_n| = |G_{n-1}| + |G_{n-2}|$, where $n \geq 2$.
- If we let $g(n) = |G_n|$ for each $n \geq 0$, we can describe g recursively as follows:

$$g(0) = 1$$
$$g(1) = 2$$
$$g(n) = g(n-1) + g(n-2) \qquad (n \geq 2)$$

Overlapping Subproblems

- This function can be directly implemented using recursion.

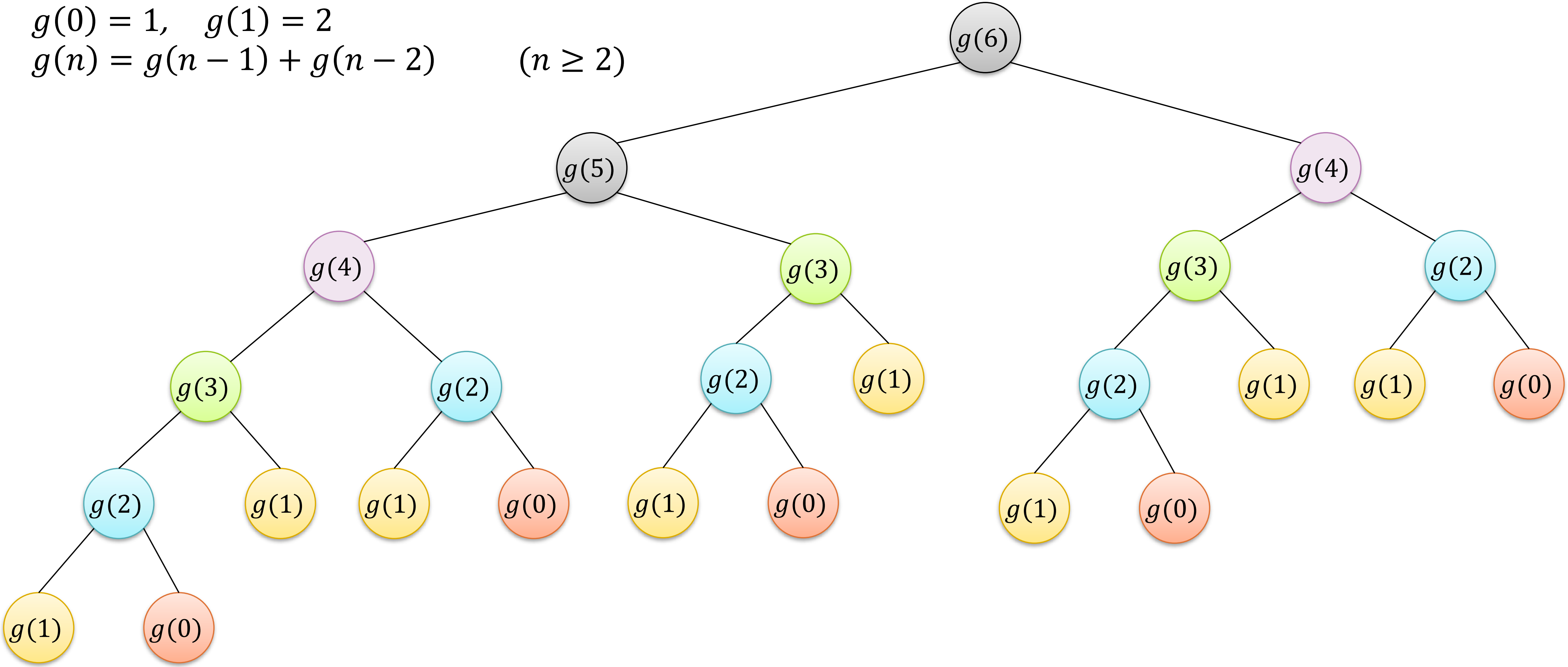
```
function g(n)
  if n==0 then
    return 1
  else if n==1 then
    return 2
  else
    return g(n-1)+g(n-2)

print(g(N))
```

Overlapping Subproblems

Let's draw a recursion tree of $g(6)$.

$g(0) = 1, \quad g(1) = 2$

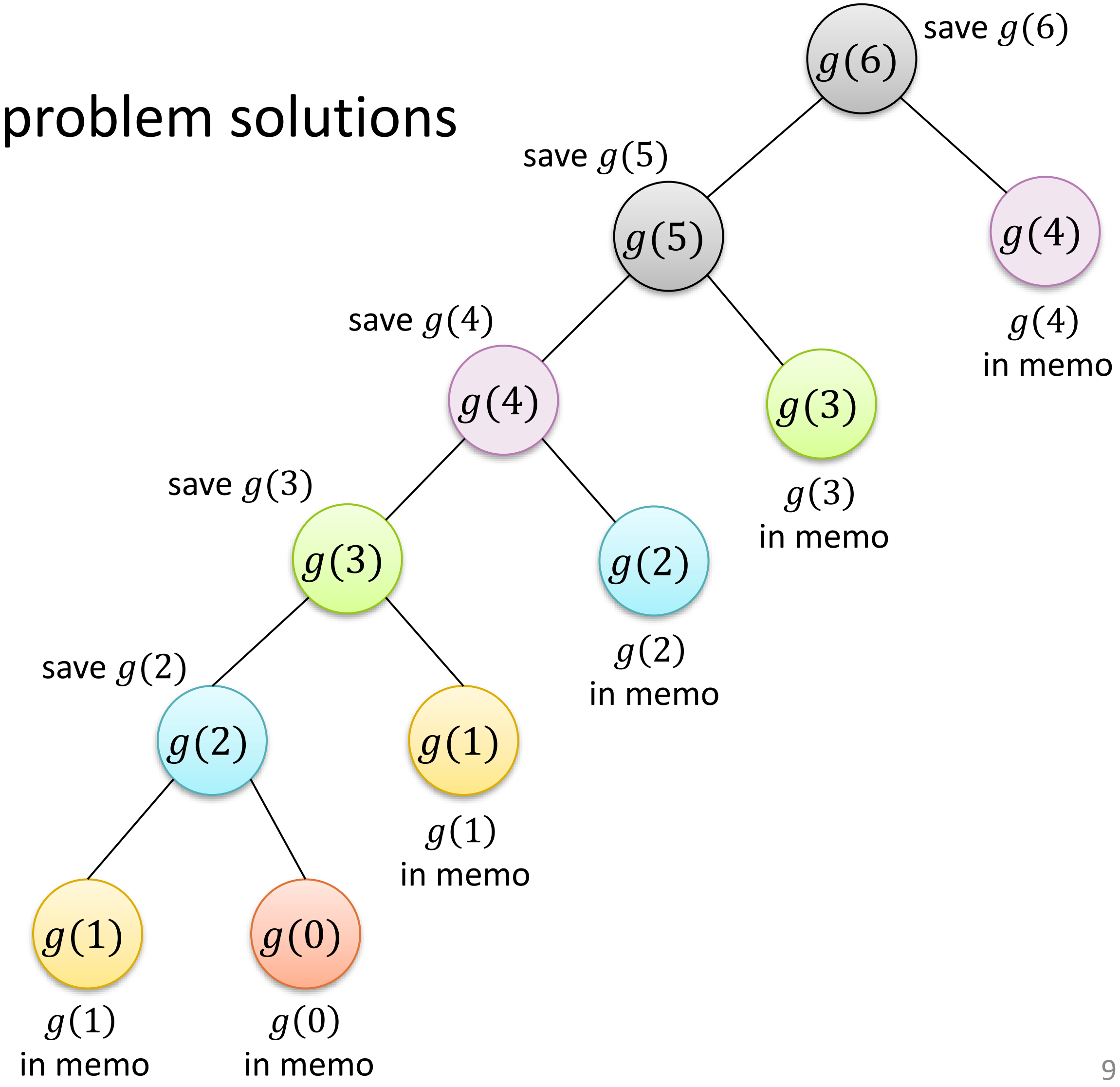
$g(n) = g(n-1) + g(n-2) \qquad (n \geq 2)$



More Efficient Implementations

- How can we improve the implementation of this function? There are 2 methods.
- **Method 1:** Memorizing the computed sub-problem solutions

```
memo = {0:1, 1:2}
function g(n)
  if n is a key in memo
    return memo[n]
  else
    memo[n] = g(n-1)+g(n-2)
    return memo[n]

print(g(N))
```



More Efficient Implementations

- **Method 2:** Perform the computation in a bottom-up manner, storing computed solutions in a table.
- For example, to computing $g(6)$, we iteratively compute

$$g(0), g(1), \dots, g(5), g(6)$$

```
g[0…N] = Array of length N+1
g[0] = 1
g[1] = 2
for i from 2 … N
   g[i] = g[i-1]+g[i-2]
print(g[N])
```

$\boldsymbol{n}$	$\boldsymbol{g(n)}$
0	1
1	2
2	$2+1=3$
3	$3+2=5$
4	$5+3=8$
5	$8+5=13$
6	$13+8=21$

More Efficient Implementations

- This method can consume more space. But in many cases, we can save space by storing only the computed solutions that will be used later on.
- In the previous example, to find $g(n)$, we only need $g(n-1)$ and $g(n-2)$. Earlier computed solutions, i.e. $g(n-3), g(n-4), \ldots$, can be forgotten.
- In the following program, $g2$ stores $g(k-2)$ and $g1$ stores $g(k-1)$.

```
if N==0 then print(1)
else if N==1 then print(2)
else
    k = 2
    g2,g1 = 1,2
    while k<=N
       k = k+1
       g2,g1 = g1,(g1+g2)
    print(g1)
```

Subset Sum

- You are given a set S of positive integers and a positive integer Y. Determine whether there is a subset of S whose sum equals to Y.
- For example, suppose $S = \{8, 6, 7, 5, 3, 10, 9\}$ and $Y = 15$.
  - $7 + 5 + 3 = 15$
  - Also, $10 + 5 = 15$ and $8 + 7 = 15$

- How about $S' = \{11, 6, 5, 1, 7, 13, 12\}$ and $Y = 15$?
  - There is no subset of S' whose sum equals 15.

Subset Sum

We are given the initial set $S_0 = \{x_1, \dots, x_N\}$ and the initial value Y_0.

Step 1. Define the function that we need to compute. For any subset S of S_0 and any positive integer Y.

$check(S, Y) = True$ if there is a subset of S whose sum equals to Y.

$check(S, Y) = False$ otherwise.

Step 2. Write a recurrence equation describing the function.

$check(S, Y) = check(S - \{x_k\}, Y) \vee check(S - \{x_k\}, Y - x_k)$ if $Y \geq x_k$

$check(S, Y) = check(S - \{x_k\}, Y)$ if $Y < x_k$

where x_k is the last element in S.

$check(\{\}, Y) = True$ if $Y = 0$

$check(\{\}, Y) = False$ if $Y \neq 0$

Subset Sum

Step 3. Simplify the arguments of the function. Notice that the first argument of $check$ is a set containing elements $x_1, \dots, x_k$ for some $k \leq N$. So, we can instead represent the first argument by the integer k such that

$$check'(k, Y) \text{ means } check(\{x_1, \dots, x_k\}, Y)$$

and

$$check'(0, Y) \text{ means } check(\{\}, Y)$$

We can thus simplify the arguments

$$check'(k, Y) = check'(k-1, Y) \vee check'(k-1, Y - x_k) \quad \text{if } Y \geq x_k$$

$$check'(k, Y) = check'(k-1, Y) \quad \text{if } Y < x_k$$

$$check'(0, Y) = True \ \text{ if } Y = 0$$

$$check'(0, Y) = False \text{ if } Y \neq 0$$

Our goal is to find $check'(N, Y_0)$, which equals to $check(S_0, Y_0)$.

Subset Sum

Let us see how we evaluate $check'(k, Y)$ in the bottom-up manner.
Suppose $S_0 = \{1,2,4,5\}$ and $Y_0 = 8$. We compute $check'(k, Y)$ where $0 \leq k \leq 4$ and $0 \leq Y \leq 8$
First, $check'(0, Y) = True$ if $Y = 0$; otherwise, $check'(0, Y) = False$.

x_k		$Y = 0$	1	2	3	4	5	6	7	8
	$k = 0$	T	F	F	F	F	F	F	F	F
1	1	T								
2	2	T								
4	3	T								
5	4	T								

Subset Sum

For $k = 1$, we have $x_k = 1$ and thus

$$check'(1, Y) = check'(0, Y) \vee check'(0, Y - 1) \qquad \text{if } Y \geq 1$$

$$check'(1, Y) = check'(0, Y) \qquad \text{if } Y < 1$$

x_k		$Y = 0$	1	2	3	4	5	6	7	8
	$k = 0$	T	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F
2	2	T								
4	3	T								
5	4	T								

Subset Sum

For $k = 2$, we have $x_k = 2$ and thus

$$check'(2, Y) = check'(1, Y) \vee check'(1, Y-2) \qquad \text{if } Y \geq 2$$

$$check'(2, Y) = check'(1, Y) \qquad \text{if } Y < 2$$

x_k		$Y = 0$	1	2	3	4	5	6	7	8
	$k = 0$	T	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F
2	2	T	T	T	T	F	F	F	F	F
4	3	T								
5	4	T								

Subset Sum

For k =3, we have $x_k = 4$ and thus

$$check'(3, Y) = check'(2, Y) \vee check'(2, Y - 4) \qquad \text{if } Y \geq 4$$

$$check'(3, Y) = check'(2, Y) \qquad \text{if } Y < 4$$

x_k		$Y = 0$	1	2	3	4	5	6	7	8
2	2	T	T	T	T	F	F	F	F	F
4	3	T	T	T	T	T	T	T	T	F
5	4	T								

Subset Sum

For k =4, we have $x_k = 5$ and thus

$$check'(4, Y) = check'(3, Y) \vee check'(3, Y - 5) \qquad \text{if } Y \geq 5$$

$$check'(4, Y) = check'(3, Y) \qquad \text{if } Y < 5$$

We have found that $check'(4{,}8) = True$.
This implies there is a subset of S_0 whose sum equals to 8.

x_k		$Y = 0$	1	2	3	4	5	6	7	8
2	2	T	T	T	T	F	F	F	F	F
4	3	T	T	T	T	T	T	T	T	F
5	4	T	T	T	T	T	T	T	T	T

Subset Sum

Which elements of S_0 add up to 8? We can find them by tracing back from $check'(4,8)$.

		$Y = 0$	1	2	3	4	5	6	7	8
x_k	$k = 0$	T	F	F	F	F	F	F	F	F
1	1	T	T	F	F	F	F	F	F	F
2	2	T	T	T	T	F	F	F	F	F
4	3	T	T	T	T	T	T	T	T	F
5	4	T	T	T	T	T	T	T	T	T

Cutting Stick

- Suppose you have a wooden stick of length N (where N is a positive integer), and you want to cut the stick into one or more pieces and sell the pieces to maximize the profit.
  - There is a condition that the length of each piece must be a positive integer.
  - You are given the selling price $P_i \geq 0$ of a stick of length i.
  - There is a cost of $C \geq 0$ Baht for each cut you make.
- For example, suppose $N = 5$, the cutting cost $C = 10$, and the selling price P_i for a piece of length i is as follows.

i	1	2	3	4	5
P_i	30	40	120	130	150

  - If we don't cut at all (just sell the entire stick of length 5), our profit will be 150 Baht.
  - If we cut into 2 pieces of length 2 and 3, our profit will be $40 + 120 - 10 = 150$ Baht.
  - If we cut into 3 pieces of length $1, 2, 2$, our profit will be $30 + 40 + 40 - 20 = 90$ Baht.
  - If we cut into 5 pieces, each of length 1, our profit will be $5 \cdot 30 - 40 = 110$ Baht.

Cutting Stick

- **Step 1.** Name the function that we would like to compute.

  Define $opt(n)$ to be the maximal profit that could be obtained from cutting and selling a stick of length n.

- **Step 2.** Write a recurrence equation for the function.
  - If the stick length $n = 1$, it cannot be cut any further. Hence, $opt(n) = P_1$.
  - Given a stick of length $n > 1$, there n options for the first cut:
    - Not cutting at all ⇒ Maximal profit is P_n.
    - Cut 1 meter from the left end ⇒ Maximal profit is $opt(1) + opt(n-1) - C$.
    - Cut 2 meters from the left end ⇒ Maximal profit is $opt(2) + opt(n-2) - C$.
    - …
    - Cut $n-1$ meters from the left end ⇒ Maximal profit is $opt(n-1) + opt(1) - C$.
  - We choose a cut that maximizes the profit. Hence, for $n > 1$,

$$opt(n) = Max(\{P_n\} \cup \{opt(k) + opt(n-k) - C \mid 1 \leq k \leq n-1\})$$

Cutting Stick

- Here is an alternative equation
- Given a stick of length $n > 1$, there are n options for the leftmost cut:
  - Not cutting at all $\Rightarrow$ Maximal profit is P_n.
  - Leftmost cut is 1 meter from the left end $\Rightarrow$ Maximal profit is $P_1 + opt(n-1) - C$.
  - Leftmost cut is 2 meter from the left end $\Rightarrow$ Maximal profit is $P_2 + opt(n-2) - C$.
  - …
  - Leftmost cut is $n-1$ meter from the left end $\Rightarrow$ Maximal profit is $P_{n-1} + opt(1) - C$.
- We choose the leftmost cut that maximizes the profit. Hence, for $n > 1$,

$$opt(n) = Max(\{P_n\} \cup \{P_k + opt(n-k) - C \mid 1 \leq k \leq n-1\})$$

- It can be shown that these two equations for $opt(n)$ are equivalent.
- Here, we shall use this latter equation. The first equation can be implemented similarly.

Cutting Stick

- Here's the full equation.

$$opt(1) = P_1$$
$$opt(n) = Max(\{P_n\} \cup \{P_k + opt(n-k) - C \mid 1 \leq k \leq n-1\}) \qquad (\mathrm{n} > 1)$$

- **Step 3.** Simplify the arguments.
  – Since the argument in our function $opt(n)$ is already very simple, we will not be simplifying it any further.
  – Given the initial length N of the stick, the possible values of the argument n is $\{1, \dots, N\}$.

Cutting Stick

- **Step 4.** Code the function.
  - **Method 1:** Recursion with memoization.

```
memo = {1:P[1]}
function opt(n)
  if n is a key in memo
    return memo[n]
  else
    max_profit = P[n]
    for k from 1 to n-1
       max_profit = max(max_profit, P[k]+opt(n-k)-C)
    memo[n] = max_profit
    return memo[n]

print(opt(N))
```

Cutting Stick

– **Method 2:** Bottom-up implementation

```
opt[1…N] = Array of length n
opt[1] = P[1]

for n = 2 to N:
  max_profit = P[n]
  for k = 1 to n-1
    max_profit = max(max_profit, P[k]+opt[n-k]-C)
  opt[n] = max_profit

print(opt[N])
```

- From both implementations, we can determine the time complexity to be $\Theta(N^2)$.

Cutting Stick

- **Example.** Suppose $N = 5$, the cutting cost $C = 10$, and the selling price P_i for a piece of length i is as follows.

i	1	2	3	4	5
$P[i]$	30	40	120	130	150

- Let us construct the table opt.

n	$opt[n]$
1	$P[1] = 30$
2	$Max\{40, 30 + 30 - 10\} = 50$
3	$Max\{120, 30 + 50 - 10, 40 + 30 - 10\} = 120$
4	$Max\{130, 30 + 120 - 10, 40 + 50 - 10, 120 + 30 - 10\} = 140$
5	$Max\{150, 30 + 140 - 10, 40 + 120 - 10, 120 + 50 - 10, 130 + 30 - 10\} = 160$

Cutting Stick

Suppose the problem also asks for the cutting positions that produces the optimal profit.

- **Step 5.** Extract the solutions.
  - While finding the optimal profit, we need to record the choices that lead to the optimal profit.

```
memo_profit = {1:P[1]}
memo_cuts = {1:[]}
function opt(n)
  if n is a key in memo_profit
    return (memo_profit[n],memo_cuts[n])
  else
    max_profit = P[n]
    max_cuts = []
    …
```

```
    for k from 1 to n-1
      (profit_nk,cuts_nk) = opt(n-k)
      if max_profit < P[k]+profit_nk-C
        max_profit = P[k]+profit_nk-C
        max_cuts = [k]+cuts_nk[n-k]
    memo_profit[n] = max_profit
    memo_cuts[n] = max_cuts
    return memo[n]
```

Cutting Stick

```
opt_profit[1…N] = Array of length n
opt_cuts[1…N] = Array of length n
opt_profit[1] = P[1]
opt_cuts[1] = []

for n = 2 to N:
  max_profit = P[n]
  max_cuts = []
  for k from 1 to n-1
    if max_profit < P[k]+opt_profit[n-k]-C
       max_profit = P[k]+opt_profit[n-k]-C
       max_cuts = [k] + opt_cuts[n-k]
  opt_profit[n] = max_profit
  opt_cuts[n] = max_cuts

print(opt_profit[N], opt_cuts[N])
```

Cutting Stick

- **Example.** Suppose $N = 5$, the cutting cost $C = 10$, and the selling price P_i for a piece of length i is as follows.

i	1	2	3	4	5
P_i	30	40	120	130	150

- Let us construct the tables opt_profit and opt_cuts.

n	$opt\_profit[n]$	$opt\_cuts[n]$
1	$P[1] = 30$	$[]$
2	$Max\{40, 30 + 30 - 10\} = 50$	$[1]$
3	$Max\{120, 30 + 50 - 10, 40 + 30 - 10\} = 120$	$[]$
4	$Max\{130, 30 + 120 - 10, 40 + 50 - 10, 120 + 30 - 10\} = 140$	$[1]$
5	$Max\{150, 30 + 140 - 10, 40 + 120 - 10, 120 + 50 - 10, 130 + 30 - 10\} = 160$	$[1, 1]$

Optimal Changes

- Suppose we have M types of stamps with values $C_1, \dots, C_M$ Bahts, where each C_i is a positive integer. If we need to pay the postage cost of N Bahts (N is a non-negative integer) using stamps, what is the least total number of stamps needed?
  - We assume that we have an unlimited supply for the stamp of each type.
- For example, suppose there are $M = 4$ types of stamps, with the following values:

t	Type 1	Type 2	Type 3	Type 4
C_t	1	2	4	5

  and the postage cost $N = 7$ Bahts.

- Some possible stamp sets with postage values of 7 Bahts.
  - 7 x Type 1 = 7 x 1 (7 stamps)
  - (1 x Type 1) + (1 x Type 2) + (1 x Type 3) = (1 x 1) + (1 x 2) + (1 x 4) (3 stamps)
  - (1 x Type 2) + (1 x Type 4) = (1 x 2) + (1 x 5) (2 stamps)

Optimal Changes

- **Step 1.** Name the function that we would like to compute.

  Define $opt(n)$ to be the minimal number of stamps with the total value of n Bahts.

- **Step 2.** Write a recurrence equation for the function.
  - If $n = 0$, we do not need any stamp. Hence, $opt(0) = 0$.
  - Suppose $n \geq 1$. We form an optimal stamp set by picking one stamp into the set. Let us consider the possible choices for the first stamp.
    - If $n \geq C_1$, we can pick the stamp of Type 1 ⇒ Minimal number of stamps is $opt(n - C_1) + 1$.
    - If $n \geq C_2$, we can pick the stamp of Type 2 ⇒ Minimal number of stamps is $opt(n - C_2) + 1$.
    - …
    - If $n \geq C_M$, we can pick the stamp of Type M ⇒ Minimal number of stamps is $opt(n - C_M) + 1$.
  - We make the choices that result in the minimal number of stamps. Hence, for $n \geq 1$,

$$opt(n) = Min\,\{opt(n - C_t) + 1 \mid n \geq C_t \;\; and\; 1 \leq t \leq M\,\}$$

Optimal Changes

- Here's the full equation.

$$opt(0) = 0$$
$$opt(n) = Min\,\{opt(n - C_t) + 1 \mid n \geq C_t \quad and\; 1 \leq t \leq M\,\} \qquad (\mathrm{n} \geq 1)$$

- **Step 3.** Simplify the arguments.
  - Since the argument in our function $opt(n)$ is already very simple, we will not be simplifying it any further.
  - Given the initial postage cost of N Bahts, the possible values of the argument n is $\{0, \dots, N\}$.

Optimal Changes

- **Step 4.** Code the function.
  - **Method 1:** Recursion with memoization.

```
memo = {0:0}
function opt(n)
  if n is a key in memo
    return memo[n]
  else
    min_stamp_count = Infinity
    for t from 1 to M
      if n >= C[t]
        min_stamp_count = min(min_stamp_count, memo[n-C[t]]+1)
    memo[n] = min_stamp_count
    return memo[n]

print(opt(N))
```

Optimal Changes

– **Method 2:** Bottom-up implementation

```
opt[0…N] = Array of length n
opt[0] = 0

for n = 1 to N:
  min_stamp_count = Infinity
  for t = 1 to M
    if n >= C[t]
        min_stamp_count = min(min_stamp_count, opt[n-C[t]]+1)
    opt[n] = min_stamp_count

print(opt[N])
```

- From both implementations, we can determine the time complexity to be $\Theta(N \cdot M)$

Optimal Changes

- **Example.** Suppose the postage cost N =7 and there are $M = 4$ stamp types with the following values:

t	Type 1	Type 2	Type 3	Type 4
C_t	1	2	4	5

- Let us construct the table opt.

n	$opt[n]$
0	0
1	$opt[1-1]+1=1$
2	$Min\{opt[2-1]+1, opt[2-2]+1\} = Min\{2,1\} = 1$
3	$Min\{opt[3-1]+1, opt[3-2]+1\} = Min\{2,2\} = 2$
4	$Min\{opt[4-1]+1, opt[4-2]+1, opt[4-4]+1\} = Min\{3,2,1\} = 1$
5	$Min\{opt[5-1]+1, opt[5-2]+1, opt[5-4]+1, opt[5-5]+1\} = Min\{2,3,2,1\} = 1$
6	$Min\{opt[6-1]+1, opt[6-2]+1, opt[6-4]+1, opt[6-5]+1\} = Min\{2,2,2,2\} = 2$
7	$Min\{opt[7-1]+1, opt[7-2]+1, opt[7-4]+1, opt[7-5]+1\} = Min\{3,2,3,2\} = 2$

1D Range Sum

- You are given a sequence of n integers, x_1, x_2, ..., x_n, say

i	1	2	3	4	5	6	7	8	9	10
x_i	37	24	7	15	-8	29	-10	9	43	23

- and m pairs of indices (a,b) where $1 \leq a \leq b \leq n$, say we have 1,000,000 pairs as follows:

(2,6) (1,7) (3,8) (9,9) ... (3,7)

- You are asked to write a program which, for each pair (a,b), compute the sum from a to b, precisely

$$\text{sum}(a,b) = x_a + x_{a+1} + \ldots + x_b.$$

- For example, given the above input, the output should be

67, 94, 42, 43, ..., 33

1D Range Sum

- A simple algorithm:

```
Pre-condition:
  - x[1…n] contains the given sequence of integers
  - a[1…m] and b[1…m] contains pairs of indices

for j = 1 … m
  sum = 0
  for i = a[j]  … b[j]
    sum = sum + x[i]
  print(sum)
```

- What is the time complexity of this algorithm in terms of n and m?

1D Range Sum

- If there are a large number of pairs (i.e. m is very large), there is a more efficient technique.
- First, compute the sum $s_i = x_1 + x_2 + \ldots + x_i$ for each $i \leq n$.
  Define s_0 to be 0.
- For example, suppose x_i are

i	1	2	3	4	5	6	7	8	9	10
x_i	37	24	7	15	-8	29	-10	9	43	23

- Then s_i are

i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	37	61	68	83	75	104	94	103	146	169

- Then the sum from a to b, denoted sum(a, b) can be calculated easily by

$$\mathrm{sum}(a, b) = s_b - s_{a-1}$$

1D Range Sum

- A revised algorithm:

```
Pre-condition:
  - x[1…n] contains the given sequence of integers
  - a[1…m] and b[1…m] contains pairs of indices

// Compute the partial sums si
s[0] = 0
for i = 1 … n
  s[i] = s[i-1] + x[i]

// Compute the sum for each pair of indices
for j = 1 … m
  sum = s[b[j]] - s[a[j]-1]
  print(sum)
```

- What is the time complexity of this algorithm in terms of n and m?

Max 1D Range Sum

- Given a sequence of n integers, x_1, x_2, ..., x_n, find a pair (a,b) of indices which maximizes the partial sum, i.e.

$$\text{sum}(a,b) \geq \text{sum}(x,y) \quad \text{for all indices } x, y \text{ where } x \leq y$$

- A straightforward implementation:

```
a = b = 1
max = x[a]

for i = 1 … n
  for j = 1 … n
    if(sum(i,j) > max)
      max = sum(i,j)
      a = i
      b = j
return (a,b)
```

- What's the time complexity of this method?

Max 1D Range Sum – Kadane's Algorithm

- There is an O(n) algorithm to solve the Max 1D Range Sum problem. The algorithm described below is attributed to Jay Kadane.

```
// Kadane's Algorithm
sum = 0
max_sum = 0

for i = 1 … n
  sum += x[i]
  max_sum = max(max_sum, sum)
  if(sum < 0) sum = 0

return max_sum
```

- Try running the above algorithm on the array x =

4, -5, 4, -3, 4, 4, -4, 4, -5

Max 1D Range Sum – Kadane's Algorithm

- Try running Kadane's algorithm on the array x = [4, -5, 4, -3, 4, 4, -4, 4, -5]

	i	1	2	3	4	5	6	7	8	9
	$x[i]$	4	-5	4	-3	4	4	-4	4	-5
sum	0	4	~~-1~~ 0	4	1	5	9	5	9	4
max_sum	0	4	4	4	4	5	9	9	9	9

2D Range Sum

- Now, instead of a sequence of integers, you are now given a n x n table of integers,

$$x_{1,1}, x_{1,2}, \ldots, x_{1,n},$$
$$x_{2,1}, x_{2,2}, \ldots, x_{2,n},$$
$$\ldots$$
$$x_{n,1}, x_{n,2}, \ldots, x_{n,n}$$

- For example, suppose $x_{i,j}$ are given below

j / i	1	2	3	4	5
1	7	-4	7	15	-8
2	3	2	-1	9	0
3	-2	2	2	1	1
4	6	4	0	1	7
5	-8	-2	-4	-8	-1

2D Range Sum

- You are then given m pairs of locations in the table, each written in the form (a, b, c, d), where (a,b) refers to a location in the table and similarly for (c, d) with the condition that $1 \leq a \leq c \leq n$ and
  $1 \leq b \leq d \leq n$.
- For each given tuple (a, b, c, d), you compute the sum from (a,b) to (c,d):

$$\text{sum}(a,b,c,d) = \sum_{i=a}^{c} \sum_{j=b}^{d} x_{i,j}$$

- For example, from the previous table,

$$\begin{aligned} \text{sum}(2,1,4,3) = \;& 3 + 2 + (-1) + \\ & -2 + 2 + 2 + \\ & 6 + 4 + 0 \qquad = 16 \end{aligned}$$

2D Range Sum

- A simple algorithm:

```
Pre-condition:
  - x[1…n, 1…n] contains the given n x n table of
    integers.
  - a[1…m],b[1…m],c[1…m],d[1…m] describe m pairs of
    locations in the table.

for k = 1 … m
  sum = 0
  for i = a[k]  … c[k]
    for j = b[k]  … d[k]
      sum = sum + x[i,j]
  print(sum)
```

- What is the time complexity of this algorithm in terms of n and m?

2D Range Sum

- We can use a similar technique as in the 1D range sum problem.
- First, compute the sum

$$s_{a,b} = \sum_{i=1}^{a} \sum_{j=1}^{b} x_{i,j} \qquad 1 \le a, b \le n$$

$$s_{0,b} = s_{a,0} = s_{0,0} = 0 \qquad 1 \le a, b \le n$$

- For example,

j / i	1	2	3	4	5
1	7	-4	7	15	-8
2	3	2	-1	9	0
3	-2	2	2	1	1
4	6	4	0	1	7
5	-8	-2	-4	-8	-1

j / i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	7	3	10		
2	0	10	8	14		
3	0	8	8	16		
4	0					
5	0					

2D Range Sum

- We can compute $s_{a,b}$ recursively:

$$s_{a,b} = s_{a-1,b} + s_{a,b-1} - s_{a-1,b-1} + x_{a,b} \qquad 1 \leq a, b \leq n$$

$$s_{0,b} = s_{a,0} = s_{0,0} = 0 \qquad 1 \leq a, b \leq n$$

j i	1	2	3	4	5
1	7	-4	7	15	-8
2	3	2	-1	9	0
3	-2	2	2	1	1
4	6	4	0	1	7
5	-8	-2	-4	-8	-1

j i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	7	3	10		
2	0	10	8	14		
3	0	8	8			
4	0					
5	0					

2D Range Sum

- Then we can compute sum(a, b, c, d) as follows

$$sum(a,b,c,d) = s_{c,d} - s_{a-1,d} - s_{c,b-1} + s_{a-1,b-1}$$

j / i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	7	3	10	25	
2	0	10	8	**14**	**38**	
3	0	8	8	**16**	**41**	
4	0					
5	0					

2D Range Sum

- A revised algorithm:

```
Pre-condition:
  - x[1…n, 1…n] contains the given n x n table of integers.
  - a[1…m], b[1…m], c[1…m], d[1…m] describe m pairs of locations in the
table.

//Initialize s[0,0], s[i,0], s[0,i] to zero
s[0,0] = 0
for i = 1 … n
   s[0,i] = 0
   s[i,0] = 0

//Compute s[i,j]
for i = 1 … n
  for j = 1 … n
    s[i,j] = s[i-1,j] + s[i,j-1] - s[i-1,j-1] + x[i,j]

//Compute the sum for each given pair of locations
for k = 1 … m
  sum = s[c[k],d[k]] - s[a[k]-1,d[k]] - s[c[k],b[k]-1] + s[a[k]-1,b[k]-1]
  print(sum)
```

- What is the time complexity of this algorithm in terms of n and m?

Max 2D Range Sum

- Given an n x n table of integers, find a pair (a, b, c, d) of locations in the table which maximizes the partial sum, i.e.

$$\text{sum}(a, b, c, d) \geq \text{sum}(x, y, x', y')$$

$$\text{for all indices } w, x, y, z \text{ where } x \leq x' \text{ and } y \leq y'$$

- A straightforward implementation:

```
a = b = c = d = 1
max = x[a,b]

for i1 = 1 … n
  for j1 = 1 … n
    for i2 = i1 … n
      for j2 = j1 … n
        if(sum(i1,j1,i2,j2) > max)
          max = sum(i1,j1,i2,j2)
          a = i1; b = j1
          c = i2; d = j2
return (a,b,c,d)
```

Subsequences

- Given a sequence $p = [x_1, x_2, \ldots, x_n]$ ($n \geq 0$), a subsequence of p is any sequence of elements in p that respects the ordering in p. Precisely, a subsequence of p is any sequence $[x_{i_1}, x_{i_2}, \ldots, x_{i_m}]$ where $m \geq 0$ and $1 \leq i_1 < i_2 < \cdots < i_m \leq n$.

- For example, suppose $p = [3,1,4,6,9,6,7]$. Then, the following are subsequences of p:

$$[3,1,9,7],\ [4,9],\ [1,4,6,6,7],\ [6],\ []$$

  But the following are not:

$$[1,3],\ [1,4,6,7,6], [8]$$

Increasing Sequences

- A sequence $p = [x_1, x_2, \ldots, x_m]$ ($m \geq 0$) of integers is said to be increasing if and only if

  $$x_1 \leq x_2 \leq \cdots \leq x_m$$

  or, equivalently, $i \leq j$ implies $x_i \leq x_j$ for all i, j where $1 \leq i, j \leq m$.

- The sequence $[1,4,6,6,7]$ is increasing, but $[3,1,9,7]$ is not.

Longest Increasing Subsequence (LIS)

- **Problem.** Given a sequence p of integers, find a longest increasing subsequence of p.
- For example, suppose $p = [3,1,4,6,9,6,7]$. The following are some increasing subsequences of p:

$$[3,9], [3,6,9], [9], [], [3,4,6,6,7], [1,4,6,6,7]$$

- It can be shown that $[3,4,6,6,7]$ and $[1,4,6,6,7]$ are longest increasing subsequences of p.
- Let us apply dynamic programming to solve this problem.
- We shall begin with a simpler problem: finding the **length** of a longest increasing subsequence of the given sequence p.

DP Algorithm for LIS – First Attempt

- **Step 1.** Define the function $opt(p)$, for any sequence p of integers, as follows

$$opt(p) \text{ is the length of a longest increasing subsequence of } p.$$

- **Step 2.** Write a recurrence equation describing the function $opt(p)$.
  - Suppose p is a long sequence and $p = p' \cdot [x]$, where x is the last integer in p.

p'	x

  - An LIS of p is the longer of the following
    a) an LIS of p' or
    b) an LIS of p' that ends with an integer no larger than x, and then appended by x.
  - We can let $opt(p)$ be the greater of the lengths of the above subsequences.
  - We can find the length of (a) from $opt(p')$. But we have no way to easily determine the length of (b) from $opt(p')$.

DP Algorithm for LIS – Second Attempt

- There seems to be no effective recurrence equations for the function opt we previously defined. We shall introduce an additional parameter to the function that allows us to define the function recursively.
- **Step 1.** Define the function $opt(p, k)$, for any sequence p of integers and $1 \leq k \leq len(p)$, as follows

  $opt(p, k)$ is the length of a longest increasing subsequence of p that ends at index k.

- **Step 2.** Write a recurrence equation describing the function $opt'(p, k)$.
  - If p is an empty sequence, then obviously $opt'(p, k) = 0$. So let us focus on the case where p is non-empty.
  - Suppose $k = 1$. Then, $opt(p, 1)$ is the length of a longest increasing subsequence of p ending at index 1. Obviously, such subsequence contains just $p[1]$. Hence, $opt(p, 1) = 1$.

DP Algorithm for LIS – Second Attempt

– Suppose $k > 1$.

1	2	3	…	$k-1$	k	…
$p[1]$	$p[2]$	$p[3]$	…	$p[k-1]$	$p[k]$	…

– An LIS of p ending at index k must be of the form $q + [p[k]]$ where q is a longest increasing subsequence of $p[1 \ldots k-1]$ that ends with some integer $\leq p[k]$.

– Hence, we can write

$$opt(p,k) = 1 + Max\,\{\, opt(p,i) \;\mid 1 \leq i < k \; and \; p[i] \leq p[k] \;\}$$

If there all elements $p[i]$ are greater than $p[k]$ (hence, the set under the Max operator above is empty), then an LIS ending at index k contains just $p[k]$. Hence, in this case $opt(p,k) = 1$.

DP Algorithm for LIS – Second Attempt

- The full equation is as follows. For any sequence p of integers and $1 \leq k \leq len(p)$,

$$opt(p, 1) = 1$$

$$opt(p, k) = 1 + Max\ \{\ opt(p, i)\ \mid 1 \leq i < k\ and\ p[i] \leq p[k]\ \}$$

if $k > 1$ and there is some $i < k$ where $p[i] \leq p[k]$

$$opt(p, k) = 1$$

if $k > 1$ and all $p[i] > p[k]$ for all $i < k$.

DP Algorithm for LIS – Second Attempt

- **Step 3.** Simplify the parameters of the recurrence equation.
  - Notice from the recurrence equation that the first parameter p remains unchanged. So, we can fix the list p (e.g. storing p in a global variable in the code) and omit it from the parameters of opt.
  - Fix a sequence p. For any integer k where $1 \leq k \leq len(p)$:

    $opt(k)$ is the length of a longest increasing subsequence of p that ends at index k
  - The recurrence equation is then simplified to the following:

$$opt(1) = 1$$

$$opt(k) = 1 + Max\,\{\, opt(i) \mid 1 \leq i < k \text{ and } p[i] \leq p[k] \,\} \quad \text{if } k > 1 \text{ and there is some } i < k \text{ where } p[i] \leq p[k]$$

$$opt(k) = 1 \quad \text{if } k > 1 \text{ and all } p[i] > p[k] \text{ for all } i < k.$$

- An LIS of p will end at some index of p, which may not be at the last index.
  The length of an LIS of p is the maximum of $opt(k)$ for all indices k. Precisely,

$$LIS(p) = Max\,\{opt(k) \mid 1 \leq k \leq len(p)\}$$

DP Algorithm for LIS – Example

Find the length of an LIS of $p = [9, 3, 2, 5, 7, 4, 8, 1]$

k	1	2	3	4	5	6	7	8
$p[k]$	9	3	2	5	7	4	8	1
$opt(k)$	1	1	1	2	3	2	4	1

The length of an LIS of p is $Max\{opt(k) \mid 1 \leq k \leq 8)\} = Max\ \{1,1,1,2,3,2,4,1\} = 4$.

Sequence Alignment

- One problem that often pops up is determining the similarity or two strings.
- For example, consider the following pairs of strings

$$u_1 = BOAT \text{ and } v_1 = BOOT$$
$$u_2 = BOAT \text{ and } v_2 = BATH$$
$$u_3 = BOAT \text{ and } v_3 = BAT$$
$$u_4 = BOAT \text{ and } v_4 = BARN$$
$$u_5 = BOAT \text{ and } v_5 = BLOAT$$

- Can you order these pairs of strings from the most similar pair to the least similar pair?

Sequence Alignment

- One way to rank the similarity of pairs of strings is to define and compute the edit distance of each string pair.
- One string distance measure, called the Levenshtein distance, counts the least number of the following editing operations to convert the first string into the second string:
  - **Substitution**: replace one character in the first string by another character
  - **Deletion**: delete one character from the first string
  - **Insertion**: insert one character into the first string
- The following are examples of shortest sequences of editing operations.

A shortest sequence of editing operations	Levenshtein's edit distance
$BO\underline{A}T \xrightarrow{Subst} BO\underline{O}T$	1
$B\underline{O}AT \xrightarrow{Delete} BAT$	1
$BOAT \xrightarrow{Insert} B\underline{L}OAT$	1
$B\underline{O}AT \xrightarrow{Delete} BAT \xrightarrow{Insert} BAT\underline{H}$	2
$B\underline{O}AT \xrightarrow{Delete} BA\underline{T} \xrightarrow{Subst} BA\underline{R} \xrightarrow{Insert} BAR\underline{N}$	3

Sequence Alignment

- One way to find the edit distance to try to "align" the two strings in order to minimize the mismatches. Such an alignment is called an optimal alignment.
- The following are examples of optimal alignments of $BOAT$ and $BARN$.

$$\frac{B\ O\ A\ T\ \square}{B\ \square\ A\ R\ N} \qquad \frac{B\ O\ A\ \square\ T}{B\ \square\ A\ R\ N}$$

- Both of these alignments have 3 mismatches, the least possible.
- We can use dynamic programming to find an optimal alignment of the given pair of strings.
- We begin with the simpler problem: finding the number of mismatches in an optimal alignment. After that we shall look at how we can construct an optimal alignment.

Sequence Alignment

- **Step 1.** Define the function. Let us define for each strings u and v (over some alphabet A):

$$opt(u, v) = \text{the number of mismatches in an optimal alignment of } u \text{ and } v.$$

  – For example, $opt(\text{"BOAT"}, \text{"BARN"}) = 3$.

- **Step 2.** Write recurrence equation defining opt.

  – Let us first consider the basis case. Suppose one of the two strings is empty. For example, u is the empty string and v is a string of length m, say $v = v_1, \dots, v_n$. Then, the optimal alignment of u and v involves inserting n characters into u. Similarly, if v is the empty string and $u = u_1, \dots, u_m$ is a string of length m, then the optimal alignment involves m deletion.

$$\frac{\square \;\; \square \;\; \cdots \;\; \square}{v_1 \;\; v_2 \;\; \cdots \;\; v_n} \qquad \frac{u_1 \;\; u_2 \;\; \cdots \;\; u_m}{\square \;\; \square \;\; \cdots \;\; \square}$$

  – We can thus write the basis case of the recurrence equation.

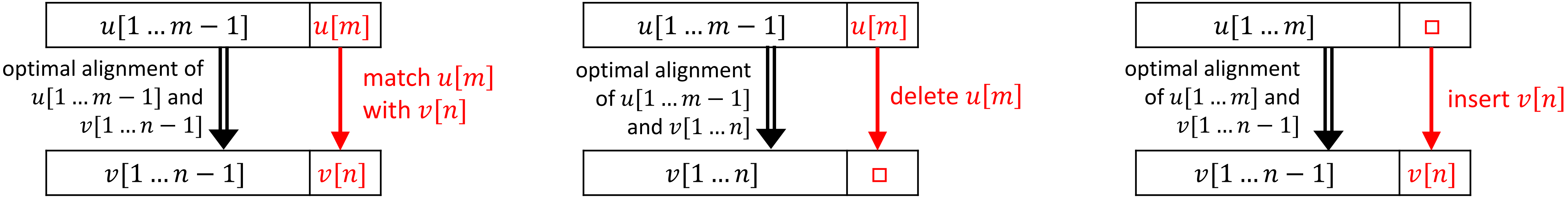
$$opt(u, \text{" "}) = |u|$$
$$opt(\text{" "}, v) = |v|$$

Note: Here, $|s|$ denotes the length of string s.

Sequence Alignment

– Next, we consider the case in which both u and v are non-empty.

– An optimal alignment of u and v must be in one of the following form:



– Thus, the number of mismatches in an optimal alignment of u and v must be the least of the numbers of mismatches in the above 3 cases. We thus have the following equation.

$$opt(u[1 \dots m], v[1 \dots n]) = Min \begin{cases} \begin{cases} opt(u[1 \dots m-1], v[1 \dots n-1]) & if\ u[m] = v[n] \\ opt(u[1 \dots m-1], v[1 \dots n-1]) + 1 & otherwise \end{cases} \\ opt(u[1 \dots m-1], v[1 \dots n]) + 1 \\ opt(u[1 \dots m-1], v[1 \dots n-1) + 1 \end{cases}$$

Sequence Alignment

- **Step 3.** Simplify the parameters.
  - Suppose the initial arguments of the function are strings u and v, respectively. The arguments of subsequent recursive calls are prefixes of u and v.
  - Thus, instead of passing around prefix strings of u and v, we can fix the initial strings u and v (e.g. storing them in global variables) and instead pass the lengths of the prefix strings.
  - The function opt is thus revised as follows: For any non-negative integers m and n where $0 \leq m \leq |u|$ and $0 \leq n \leq |v|$,

    $opt(m, n) =$ the number of mismatches in an optimal alignment of $u[1 \dots m]$ and $v[1 \dots n]$.
  - The recurrence equation defining opt is as follows.

$$opt(m, 0) = m \qquad\qquad opt(0, n) = n$$

$$opt(m, n) = Min \begin{cases} \begin{cases} opt(m-1, n-1) & if\ u[m] = v[n] \\ opt(m-1, n-1) + 1 & otherwise \end{cases} \\ opt(m-1, n) + 1 \\ opt(m, n-1) + 1 \end{cases} \qquad \text{if } m, n > 0$$

Sequence Alignment

$$opt(m,0) = m \qquad opt(0,n) = n$$

$$opt(m,n) = Min\begin{cases} \begin{cases} opt(m-1,n-1) & if\ u[m] = v[n] \\ opt(m-1,n-1)+1 & otherwise \end{cases} \\ opt(m-1,n)+1 \\ opt(m,n-1)+1 \end{cases}$$

- **Example.** Let $u = SPEAK$ and $v = PARK$.

		“ ”	P	A	R	K
		0	1	2	3	4
“ ”	0	0	1	2	3	4
S	1	1	1	2	3	4
P	2	2				
E	3	3				
A	4	4				
K	5	5				

Sequence Alignment

$$opt(m,0) = m \qquad opt(0,n) = n$$

$$opt(m,n) = Min \begin{cases} \begin{cases} opt(m-1,n-1) & if\ u[m] = v[n] \\ opt(m-1,n-1)+1 & otherwise \end{cases} \\ opt(m-1,n)+1 \\ opt(m,n-1)+1 \end{cases}$$

- **Example.** Let $u = SPEAK$ and $v = PARK$.

		$“\,”$	P	A	R	K
		0	**1**	**2**	**3**	**4**
$“\,”$	**0**	0	1	2	3	4
S	**1**	1	1	2	3	4
P	**2**	2	1	2	3	4
E	**3**	3				
A	**4**	4				
K	**5**	5				

Sequence Alignment

$$opt(m,0) = m \qquad opt(0,n) = n$$

$$opt(m,n) = Min \begin{cases} \begin{cases} opt(m-1,n-1) & if\ u[m] = v[n] \\ opt(m-1,n-1)+1 & otherwise \end{cases} \\ opt(m-1,n)+1 \\ opt(m,n-1)+1 \end{cases}$$

- Example. Let $u = SPEAK$ and $v = PARK$.

		“ ”	P	A	R	K
		0	1	2	3	4
“ ”	0	0	1	2	3	4
S	1	1	1	2	3	4
P	2	2	1	2	3	4
E	3	3	2	2	3	4
A	4	4				
K	5	5				

Sequence Alignment

$$opt(m,0) = m \qquad opt(0,n) = n$$

$$opt(m,n) = Min\begin{cases}\begin{cases} opt(m-1,n-1) & if\ u[m] = v[n] \\ opt(m-1,n-1)+1 & otherwise\end{cases} \\ opt(m-1,n)+1 \\ opt(m,n-1)+1\end{cases}$$

- **Example.** Let $u = SPEAK$ and $v = PARK$.

		" "	P	A	R	K
		0	**1**	**2**	**3**	**4**
" "	**0**	0	1	2	3	4
S	**1**	1	1	2	3	4
P	**2**	2	1	2	3	4
E	**3**	3	2	2	3	4
A	**4**	4	3	2	3	4
K	**5**	5				

Sequence Alignment

$$opt(m,0) = m \qquad opt(0,n) = n$$

$$opt(m,n) = Min\begin{cases}\begin{cases} opt(m-1,n-1) & if\ u[m] = v[n] \\ opt(m-1,n-1)+1 & otherwise\end{cases} \\ opt(m-1,n)+1 \\ opt(m,n-1)+1\end{cases}$$

- **Example.** Let $u = SPEAK$ and $v = PARK$.

		" "	P	A	R	K
		0	1	2	3	4
" "	0	0	1	2	3	4
S	1	1	1	2	3	4
P	2	2	1	2	3	4
E	3	3	2	2	3	4
A	4	4	3	2	3	4
K	5	5	4	3	3	3

Sequence Alignment

- **Example.** Let $u = SPEAK$ and $v = PARK$.

		“ ”	P	A	R	K
		0	**1**	**2**	**3**	**4**
“ ”	**0**	0	1	2	3	4
S	**1**	1	1	2	3	4
P	**2**	2	1	2	3	4
E	**3**	3	2	2	3	4
A	**4**	4	3	2	3	4
K	**5**	5	4	3	3	3

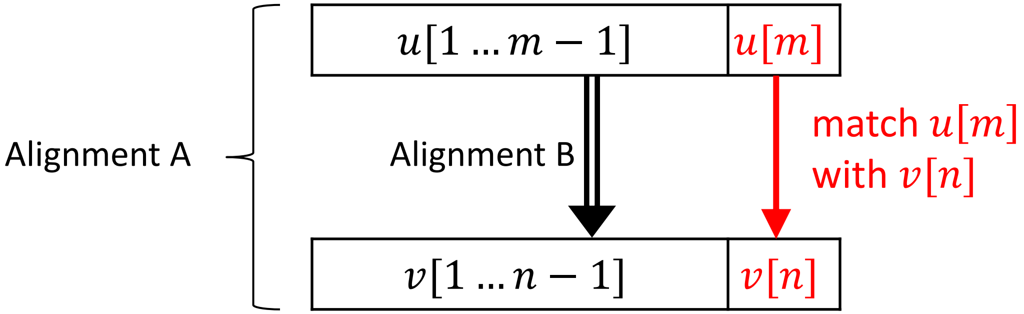
Sequence Alignment

- **Example.** Let $u = SPEAK$ and $v = PARK$.
- We have thus found that $opt(5{,}4) = 3$, which implies that the edit distance of u and v is 3.
- By tracing how the value $opt(5{,}4) = 3$ is obtained from $opt(m, n)$ for smaller arguments m and n, we can deduce that there are two optimal alignments of u and v:

$$\frac{S \quad P \quad E \quad A \quad K}{\square \quad P \quad A \quad R \quad K} \qquad\qquad \frac{S \quad P \quad E \quad A \quad \square \quad K}{\square \quad P \quad \square \quad A \quad R \quad K}$$

Characteristics of Problems where DP is Effective

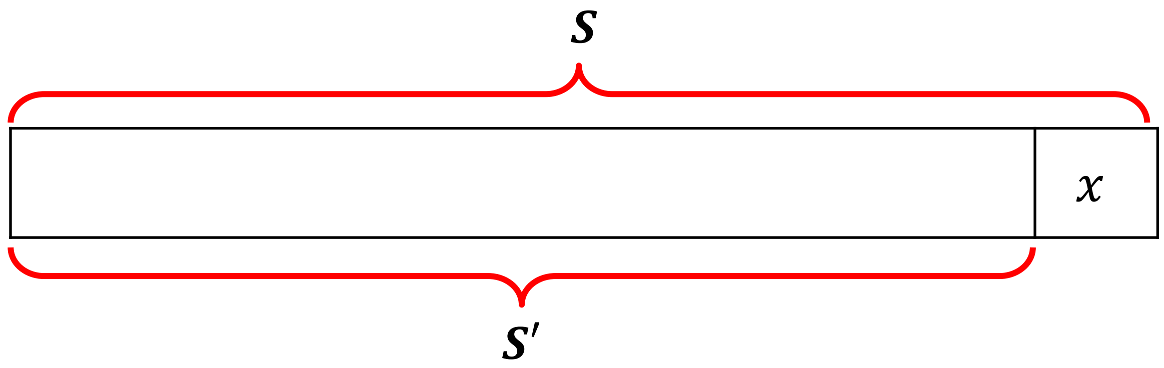
- Dynamic programming is effective for problems that possesses two key properties: **optimal substructure** and **overlapping subproblem**.
- **Optimal substructure:** A problem P can be decomposed into smaller subproblems $P_1, \dots, P_k$ such that a solution of P is composed of solutions of $P_1, \dots, P_k$.
- In the sequence alignment problem, recall that an optimal alignment of 2 strings is one with the least number of mismatches. Suppose the following alignment A is an optimal alignment of strings u and v.



- Then, alignment B must be an optimal alignment for the subproblem $u[1 \dots m-1]$ and $v[1 \dots n-1]$. Why? Because if B were <u>not</u> optimal, it would mean there is another alignment, say B', with fewer mismatches. By replacing alignment B with B', we would obtain an alignment of u and v with fewer mismatches than A, contradicting the assumption that A is optimal.

Characteristics of Problems where DP is Effective

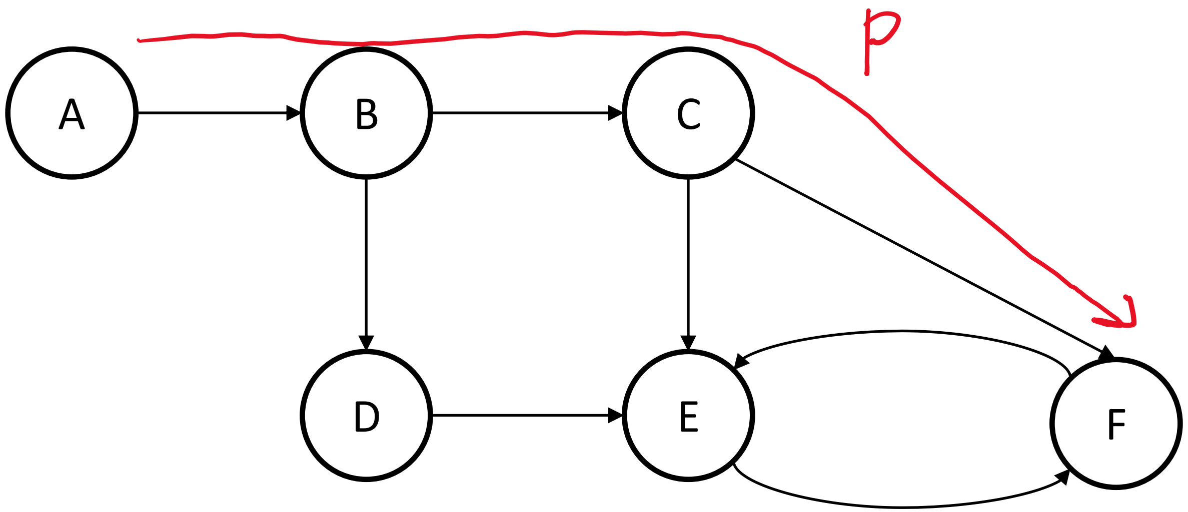
- In the coin-change problem, we would like to find a **minimal set of stamps** for the given postage cost N. Suppose $N > 0$ and S is a **minimal** set of stamps with the value of N. Since $N > 0$, S must contain at least one stamp. Suppose $S = S' \cup \{x\}$, where x is a stamp. Suppose the value of stamp x is $C(x)$ and the value of S' is $N - C(x)$.



- S' must be a minimal set of stamps with the value of $N - C(x)$. Why? Because if S' were not smallest, hence, there were a smaller set of stamps, say S'', with the value $N - C(x)$. Then, $S'' \cup \{x\}$ would be smaller than S but also have the value N. This contradicts the assumption that S is a minimal set of stamps with the value N.

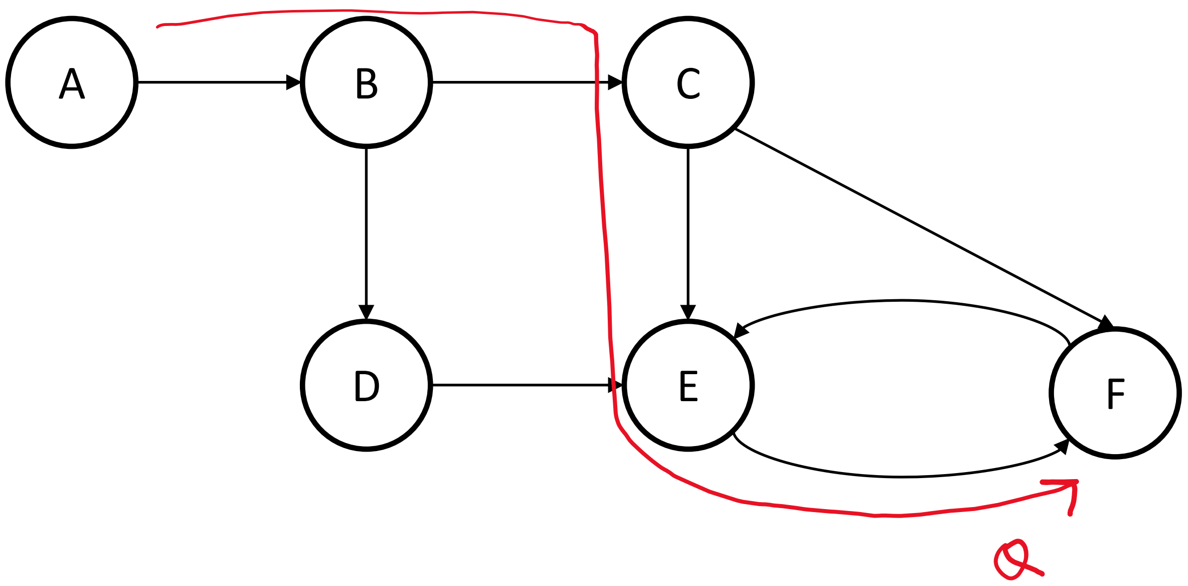
Characteristics of Problems where DP is Effective

- The **shortest-path problem** also has the optimal substructure property.
- From the graph below, a shortest path from A to F is $P = A \rightarrow B \rightarrow C \rightarrow F$. The path length is 3.
- If we look at the subpath $A \rightarrow B \rightarrow C$, this subpath must be a shortest path from A to C. Why? Because if there were a shorter path from A to C, we could replace the subpath $A \rightarrow B \rightarrow C$ in P by that shorter path. The new path would be shorter than P, contradicting the assumption that P is a shortest path from A to F.



Characteristics of Problems where DP is Effective

- On the other hand, the **longest-path problem**, which involves finding a **longest simple path** (i.e. a longest path that does not visit any node more than once) fails the optimal substructure property.
- From the graph below, a longest simple path from A to F is $Q = A \rightarrow B \rightarrow C \rightarrow E \rightarrow F$. The path length is 4.
- If this problem were to satisfy the optimal structure property, the subpath $A \rightarrow B \rightarrow C \rightarrow E$ must have been a longest simple path from A to E. But it is not! Path $A \rightarrow B \rightarrow C \rightarrow F \rightarrow E$ is actually a longer simple path from A to E. Unfortunately, this latter path cannot be extended to become a longer simple path from A to F (because adding F to this path will result in a non-simple path).



Characteristics of Problems where DP is Effective

- **Overlapping subproblems:** After breaking a problem P into subproblems $P_1, \dots, P_k$, some of these subproblems are the same or overlap, i.e. having a common sub-subproblem.
- As we previously studied, the use of memoization or bottom-up computation when implementing DP can significantly speed up the computation over the naïve (top-down) recursive implementation.
- But for a problem which does not have this overlapping subproblem property, DP is no better than the naïve recursive implementation.
- For example, the problems of **sorting** an array of distinct integers or **searching** for an element in an array of distinct elements can be reduced to smaller subproblems (i.e. sorting or searching smaller arrays) but the subproblems are different and do not overlap.