Lecture 4: Dynamic Programming

Yury Makarychev

Puzzle

- We are given n digits: $a_1 a_2 \ldots a_n$.
- Need to group the digits into one-digit and two-digit numbers so that their sum is as large as possible.

Example:

$$19118237281$$

Exponentially many feasible solutions:

$$19 + 11 + 8 + 2 + 37 + 28 + 1,\ 19 + 1 + 18 + 23 + 72 + 81, \ldots$$

Optimal solution:

$$1 + 91 + 1 + 82 + 3 + 72 + 81 = 331$$

Case 1

Assume that we know that the first case holds.

$f(n-1) + 1$

Can we find the solution?

- Solve the problem for digits $a_1, \ldots, a_{n-1}$ and add a_n.

Let $f(i)$ be the optimal solution for digits $a_1, \ldots, a_i$. Then

$$f(n) = f(n-1) + a_n$$

Case 1: Claim

- We can always find a solution of value

$$f(n-1) + a_n$$

Thus,

$$f(n) \geq f(n-1) + a_n$$

- If we are in case 1,

$$f(n-1) \geq f(n) - a_n$$

Proof: …

191182033781

$f(n-1)$ + 1

Case 1: Proof of the claim

➢ If we are in case 1,

$$f(n-1) \geq f(n) - a_n$$

Proof:

- Consider optimal solution for n of value $f(n)$.
- We are given that the last group consists of single digit a_n.
- Remove it.
- We get a feasible solution for $n-1$ of value $f(n) - a_n$. Thus

$$f(n-1) \geq f(n) - a_n$$

as required.

191182033781

$f(n-1)$ + 1

Case 1: Claim

Conclusion:

$f(n) = f(n-1) + a_n$ if we are in case 1

$f(n) \geq f(n-1) + a_n$ always

191182033781

$f(n-2)$ + 81

Case 2

Assume that we know that the second case holds.

$f(n-2)$ + 81

Can we find the solution?

- Solve the problem for digits $a_1, \ldots, a_{n-2}$ and add number $\overline{a_{n-1}a_n}$.

$$f(n) = f(n-2) + \overline{a_{n-1}a_n}$$ if we are in case 2

$$f(n) \geq f(n-2) + \overline{a_{n-1}a_n}$$ always

here $\overline{a_{n-1}a_n} = 10\, a_{n-1} + a_n$

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

Plan:

- Use this formula for $f(n)$
- Recursively compute $f(n-1)$ and $f(n-2)$
- Any obstacles?

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

Our algorithm doesn't know whether case 1 or 2 holds!

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

… but we know that

$$\left.\begin{array}{l} f(n) \geq f(n-1) + a_n \\ f(n) \geq f(n-2) + \overline{a_{n-1}a_n} \end{array}\right\} \Rightarrow f(n) \geq \max(f(n-1) + a_n, f(n-2) + \overline{a_{n-1}a_n})$$

Cases 1 & 2

- Proved

$$f(n) = \max(f(n-1) + a_n, f(n-2) + \overline{a_{n-1}a_n})$$

Recurrence

- Proved a recurrence formula

$$f(i) = \max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1}a_i})$$

- Can we compute $f(n)$ recursively?

function $f(i)$

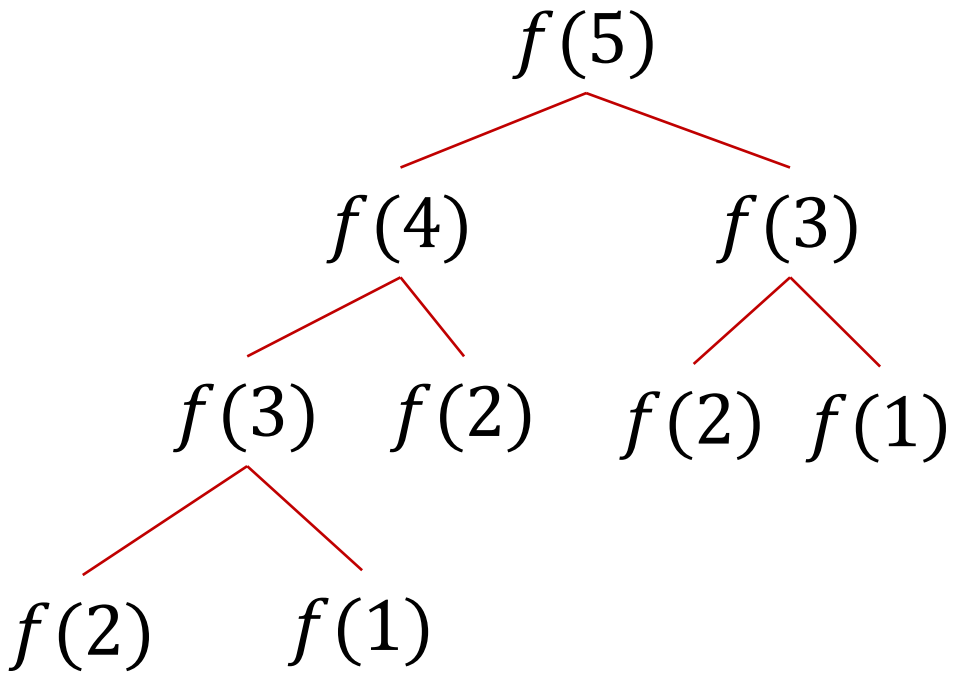
if $i = 1$, return a_1

if $i = 2$, return $\overline{a_1 a_2}$

if $i > 2$, return $\max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1}a_i})$

- This solution works correctly but it is very slow.

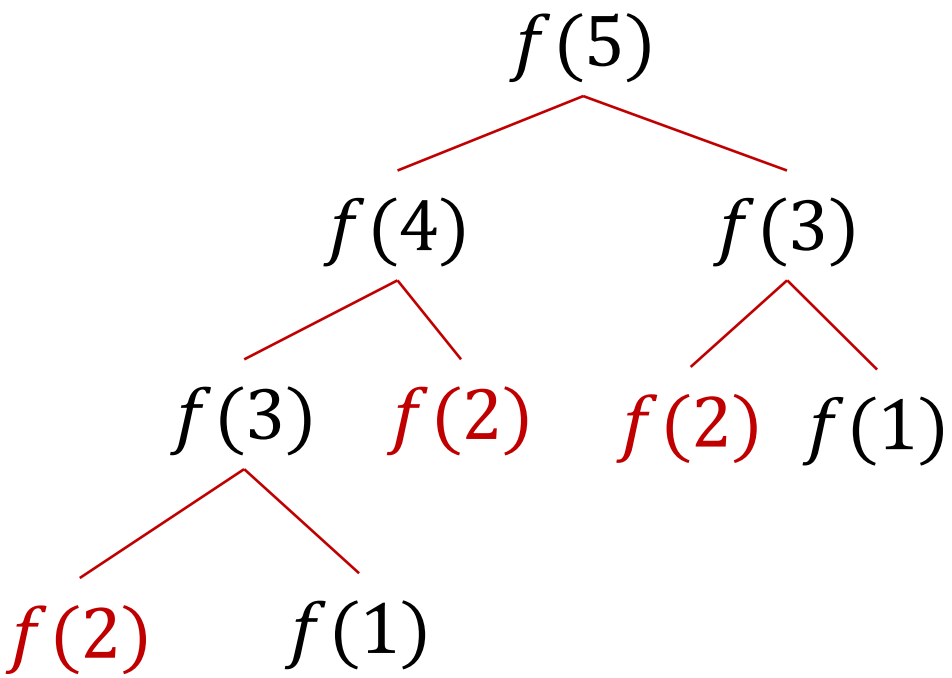
Recursion



To compute $f(5)$ we make 9 recursive calls

i	#calls
1	1
2	1
3	3
4	5
5	9
6	15
7	25
8	41
9	67
10	109
11	177
grows exponentially	

Recursion



To compute $f(5)$ we make 9 recursive calls

- compute $f(3)$ twice times
- compute $f(2)$ three times
- compute $f(1)$ twice times

… many more times when we compute $f(i)$ for large i

This is wasteful!

Memoization

- Store computed values of $f(i)$ in a table. Don't recompute them!

create a table $T[1:n]$; mark all entries as "non-initialized"

function $f(i)$

if $T[i]$ is initialized, return $T[i]$

if $i = 1$, result $= a_1$

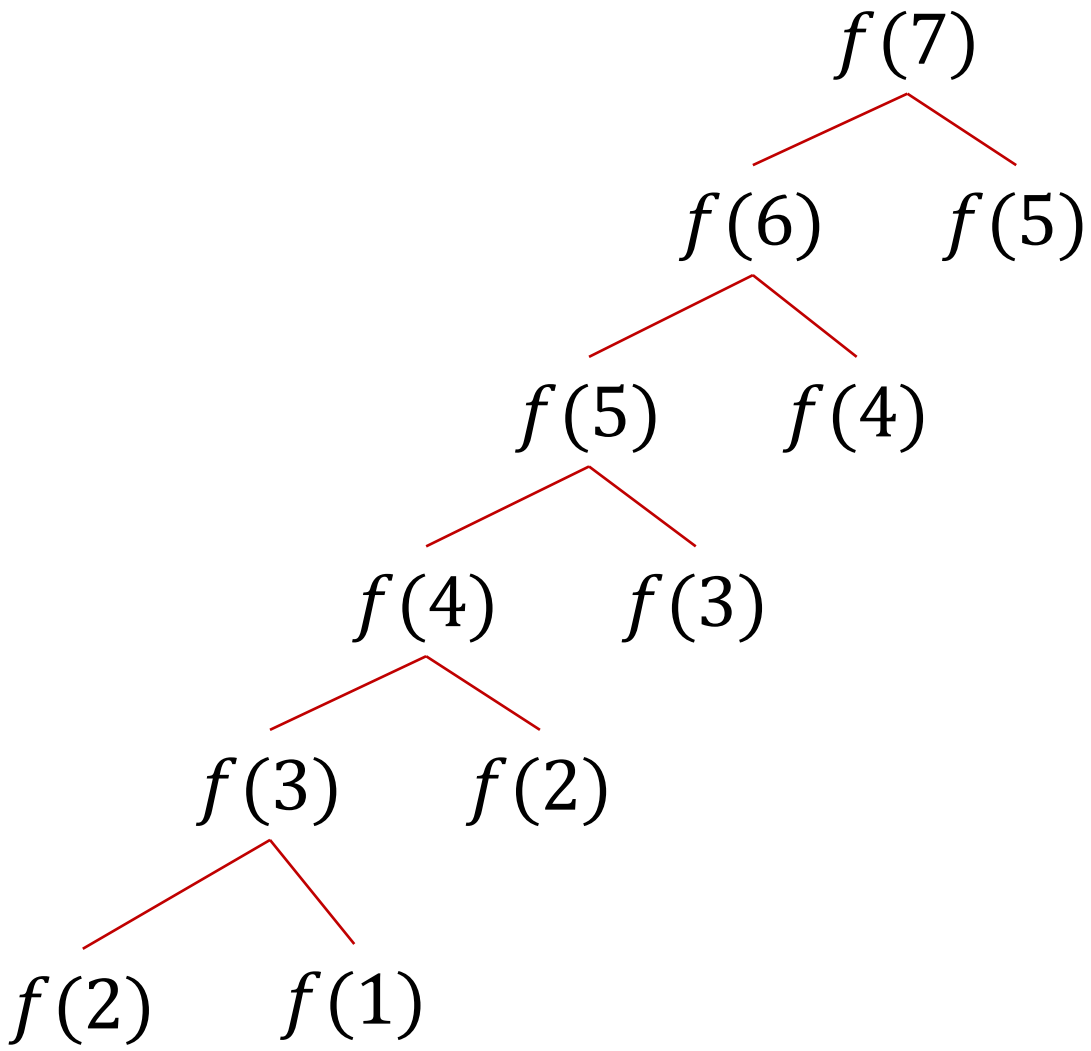
else if $i = 2$, result $= \overline{a_1 a_2}$

else if $i > 2$, result $= \max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1} a_i})$

$T[i] =$ result

return result

Memoization



i	#calls now	#calls prev.
1	1	1
2	1	1
3	3	3
4	5	5
5	7	9
6	9	15
7	11	25
8	13	41
9	15	67
10	17	109
11	19	177

linear vs exponential growth

To be continued

Unrolling the Recursion

create table $T[1:n]$
$T[1] = a_1$
$T[2] = 10 * a_1 + a_2$

for $i = 3$ to n do
    $T[i] = \max(T(i-1) + a_i, T(i-2) + \overline{a_{i-1}a_i})$

return $T[n]$

Running time is $O(n)$.

Example: 19111823

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	
1	1		1		
2	9		19		
3	1	91	92	$\max(19+1, 1+91)$	
4	1	11			
5	1	11			
6	8	18			
7	2	82			
8	3	23			

Example: 19111823

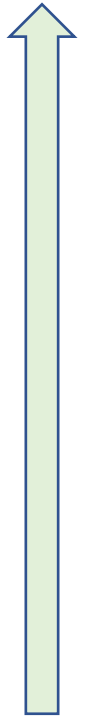
i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	
1	1		1		
2	9		19		
3	1	91	92	$\max(19+1, 1+91)$	
4	1	11	93	$\max(92+1, 19+11)$	
5	1	11	103	$\max(93+1, 92+11)$	
6	8	18	111	$\max(103+8, 93+18)$	
7	2	82	185	$\max(111+2, 103+82)$	
8	3	23	188	$\max(185+3, 111+23)$	

Example: 19111823. Finding solution

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		1
2	9	19	19		19
3	1	91	92	$\max(19+1, 1+91)$	$1+91$
4	1	11	93	$\max(92+1, 19+11)$	$1+91+1$
5	1	11	103	$\max(93+1, 92+11)$	$(1+91)+11$
6	8	18	111	$\max(103+8, 93+18)$	$1+91+11+8$ or $1+91+1+18$
7	2	82	185	$\max(111+2, 103+82)$	$1+91+11+82$
8	3	23	188	$\max(185+3, 111+23)$	$1+91+11+82+3$

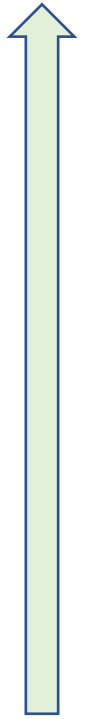
Example: 19111823. Backtracking

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		
2	9	19	19		
3	1	91	92	$\max(19+1, 1+91)$	
4	1	11	93	$\max(92+1, 19+11)$	
5	1	11	103	$\max(93+1, 92+11)$	
6	8	18	111	$\max(103+8, 93+18)$	
7	2	82	185	$\max(111+2, 103+82)$	
8	3	23	188	$\max(185+3, 111+23)$	$(expr.\ for\ 7)+3$



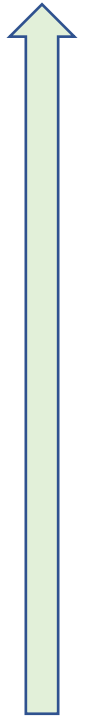
Example: 19111823. Backtracking

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		
2	9	19	19		
3	1	91	92	$\max(19+1, 1+91)$	
4	1	11	93	$\max(92+1, 19+11)$	
5	1	11	103	$\max(93+1, 92+11)$	
6	8	18	111	$\max(103+8, 93+18)$	
7	2	82	185	$\max(111+2, 103+82)$	$(expr.for\ 5) + 82 + 3$
8	3	23	188	$\max(185+3, 111+23)$	~~$(expr.for\ 7) + 3$~~



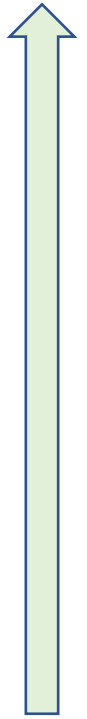
Example: 19111823. Backtracking

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		
2	9	19	19		
3	1	91	92	$\max(19+1, 1+91)$	
4	1	11	93	$\max(92+1, 19+11)$	
5	1	11	103	$\max(93+1, 92+11)$	$(expr.for\ 3) + 11 + 82 + 3$
6	8	18	111	$\max(103+8, 93+18)$	
7	2	82	185	$\max(111+2, 103+82)$	~~$(expr.for\ 5) + 82 + 3$~~
8	3	23	188	$\max(185+3, 111+23)$	~~$(expr.for\ 7) + 3$~~



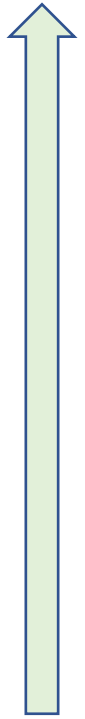
Example: 19111823. Backtracking

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		
2	9	19	19		
3	1	91	92	$\max(19+1, 1+91)$	$(expr.for\ 1) + 91 + 11 + 82 + 3$
4	1	11	93	$\max(92+1, 19+11)$	
5	1	11	103	$\max(93+1, 92+11)$	~~$(expr.for\ 3) + 11 + 82 + 3$~~
6	8	18	111	$\max(103+8, 93+18)$	
7	2	82	185	$\max(111+2, 103+82)$	~~$(expr.for\ 5) + 82 + 3$~~
8	3	23	188	$\max(185+3, 111+23)$	~~$(expr.for\ 7) + 3$~~



Example: 19111823. Backtracking

i	a_i	$\overline{a_{i-1}a_i}$	$T[i]$	formula	solution for this i
1	1		1		$1 + 91 + 11 + 82 + 3$
2	9	19	19		
3	1	91	92	$\max(19 + 1, 1 + 91)$	~~$(expr.\ for\ 1) + 91 + 11 + 82 + 3$~~
4	1	11	93	$\max(92 + 1, 19 + 11)$	
5	1	11	103	$\max(93 + 1, 92 + 11)$	~~$(expr.\ for\ 3) + 11 + 82 + 3$~~
6	8	18	111	$\max(103 + 8, 93 + 18)$	
7	2	82	185	$\max(111 + 2, 103 + 82)$	~~$(expr.\ for\ 5) + 82 + 3$~~
8	3	23	188	$\max(185 + 3, 111 + 23)$	~~$(expr.\ for\ 7) + 3$~~



Don't have to store old entries

In this case, we don't need to store old table entries.

To compute the optimal value, we can do the following:

$A = a_1$
$B = 10 * a_1 + a_2$
for $i = 3$ to n do
    $C = \max(B + a_i, A + \overline{a_{i-1}a_i})$
    $A = B$
    $B = C$
return B

Can't use this trick with backtracking.

Dynamic Programming (Approach)

- Define a set of subproblems: starting with "trivial" and ending with the given instance:

  $a_1; a_1a_2; \ldots; a_1a_2 \ldots a_n$

- Create a table T that stores the values/costs of the subproblems.
- Find a recurrence that expresses the value of an instance in terms of the values of simpler subproblems:

$$T[i] = \max(T[i-1] + a_i, T[i-2] + \overline{a_{i-1}a_i})$$

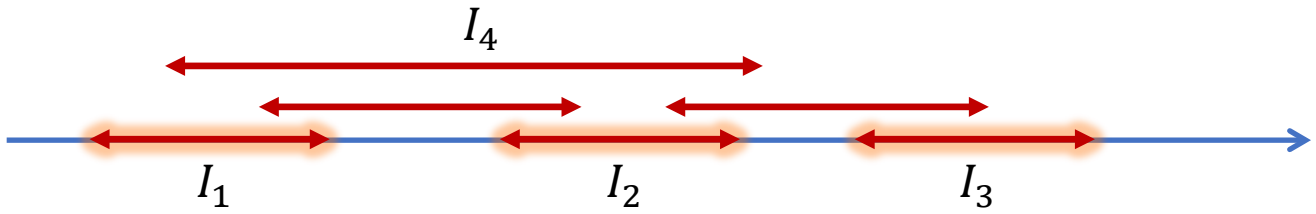
Dynamic Programming (Algorithm)

- Initialization. Fill out the values of "trivial" subproblems: $T[1] = a_1; T[2] = \overline{a_1 a_2}$
- Fill out this table:
    1. Use recursion with memorization, or
    2. Fill entries of T one-by-one, starting at the simplest entries and going toward more difficult ones.
- Keep track of the solution or use backtracking to find an optimal solution

Replit: https://replit.com/teams/join/mtwwrmwwjognndiwqjstrvvbqydjxyqp-ttic-algorithms-2024

Weighted Job/Interval Scheduling

- Given a set of jobs $I_1 = (s_1, t_1), \dots, I_n = (s_n, t_n)$ on the real line.
- Each job I_i has weight $w_i > 0$.
- A schedule is feasible, if all jobs are compatible (don't overlap)
- Find a feasible schedule $S \subset \{1, \dots, n\}$ of maximum weight $\sum_{i \in S} w_i$



Weighted Job/Interval Scheduling

- Define subproblems

Sort all jobs by their finish time t_i: $t_1 \leq t_2 \leq \cdots \leq t_n$

Let $\mathcal{P}_k$ be the instance with jobs $I_1, \ldots, I_k$.

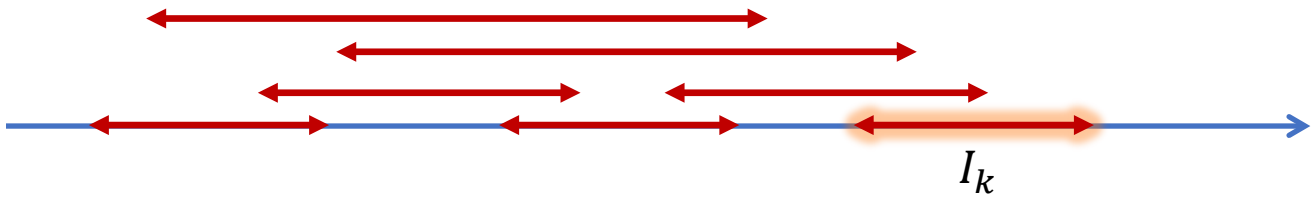
- Table

$T[0:n]$. Let $T[k]$ be value of the optimal solution for $\mathcal{P}_k$

- Initialization

$T[0] = 0$

Recurrence

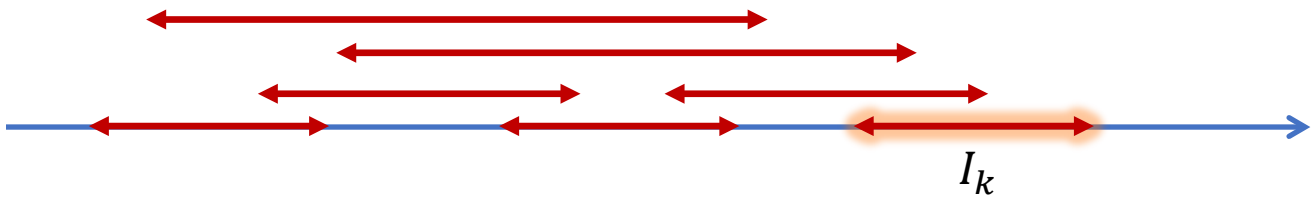


Consider jobs $I_1, \ldots, I_k$. Assume that we know an optimal schedule S^*.

There are two options

1. Job k is not in the optimal schedule: $k \notin S^*$
2. Job k is in the optimal schedule: $k \in S^*$

Recurrence: $k \notin S^*$



1. Job k is not in the optimal schedule: $k \notin S^*$

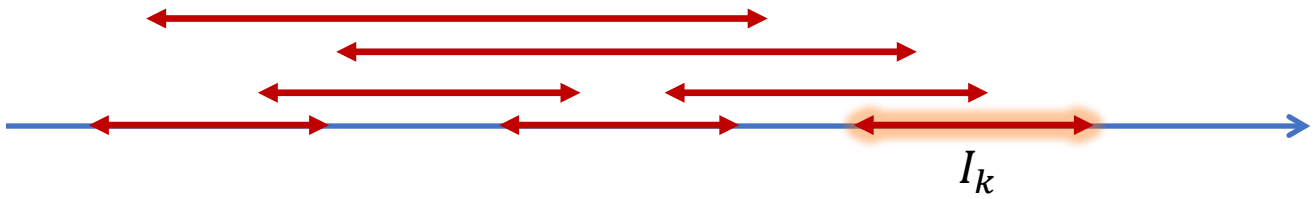
- (in case 1) S^* is also a feasible solution for $I_1, \ldots, I_{k-1}$

$$T[k-1] \geq T[k]$$

- (always) every schedule for $I_1, \ldots, I_{k-1}$ is also a schedule for $I_1, \ldots, I_k$

$$T[k] \geq T[k-1]$$

Recurrence: $k \notin S^*$



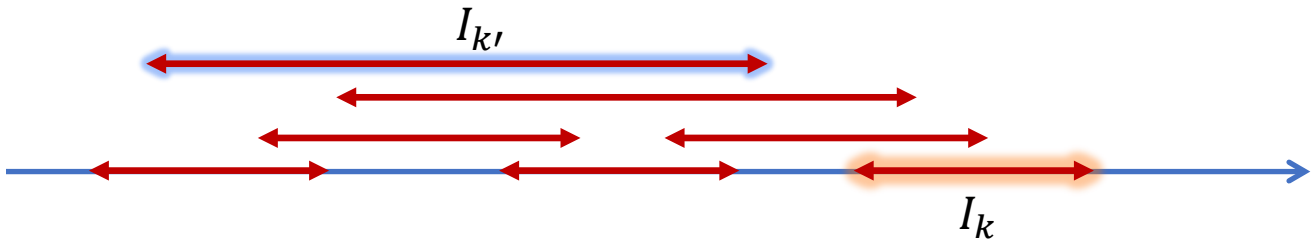
1. Job k is not in the optimal schedule: $k \notin S^*$

- (in case 1) $T[k] = T[k-1]$
- (always) $T[k] \geq T[k-1]$

Recurrence: $k \in S^*$

2. Job k is in the optimal schedule: $j \in S^*$

Let k' be the last job that lie to the left of I_k:



$$t_1 \leq t_2 \leq \cdots \leq t_{k'} \leq s_k < t_{k'+1} \leq \cdots \leq t_{k-1}$$

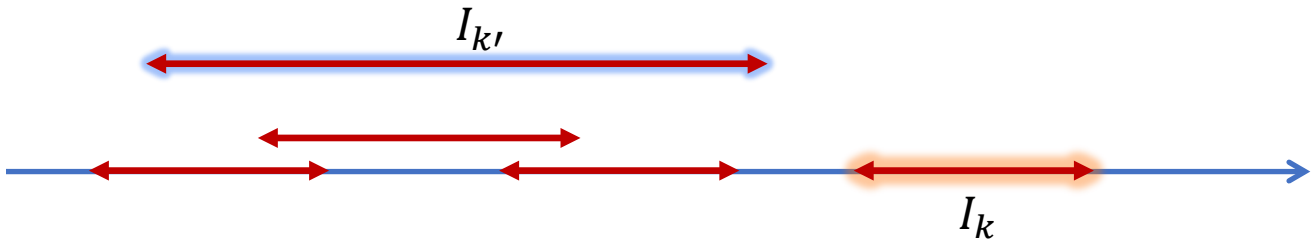
Jobs $I_1, \ldots, I_{k'}$ are compatible with I_k.

Jobs $I_{k'+1}, \ldots, I_{k-1}$ are not compatible with I_k.

Recurrence: $k \in S^*$

2. Job k is in the optimal schedule: $j \in S^*$

Let k' be the last job that lie to the left of I_k:



$$t_1 \leq t_2 \leq \cdots \leq t_{k'} \leq s_k < t_{k'+1} \leq \cdots \leq t_{k-1}$$

- $S^* \setminus \{k\}$ is a valid schedule for $\mathcal{P}_{k'}$.
- If S' is a valid solution for $\mathcal{P}_{k'}$ then $S' \cup \{k\}$ is a valid solution for $\mathcal{P}_k$

Recurrence: $k \in S^*$

- $S^* \setminus \{k\}$ is a valid schedule for $\mathcal{P}_{k'}$.

$$T[k'] \geq T[k] - w_k \qquad \text{(in case 2)}$$

- If S' is a valid solution for $\mathcal{P}_{k'}$ then $S' \cup \{k\}$ is a valid solution for $\mathcal{P}_k$

$$T[k] \geq T[k'] + w_k \qquad \text{(always)}$$

Recurrence: Summary

- $T[k] \geq \max(T[k-1], T[k'] + w_k)$
- $T[k]$ is either $T[k-1]$ or $T[k'] + w_k$

Thus:

$$T[k] = \max(T[k-1], T[k'] + w_k)$$

Two ways to construct a solution

- We can construct a solution for $I_1, \ldots, I_k$ in two ways

1) Simply take a solution for $I_1, \ldots, I_{k-1}$, or

2) Take a solution for $I_1, \ldots, I_{k'}$ and add job I_k.

- Either way, we will get a feasible solution of value

$$T[k-1] \qquad \text{or} \qquad T[k'] + w_k$$

- We choose the better of the two ways!
- The optimal solution is also obtained in one of these two ways.
  - since it is an optimal solution, it makes the same choice!

Two ways to construct a solution

- We can construct a solution for $I_1, \ldots, I_k$ in two ways

1) Simply take a solution for $I_1, \ldots, I_{k-1}$, or

2) Take a solution for $I_1, \ldots, I_{k'}$ and add job I_k.

- Either way, we will get a feasible solution of value

$$T[k-1] \qquad \text{or} \qquad T[k'] + w_k$$

- We choose the better of the two ways!
- The optimal solution is also obtained in one of these two ways.
  - since it is an optimal solution, it makes the same choice!

Two ways to construct a solution

- We can construct a solution for $I_1, \ldots, I_k$ in two ways

1) Simply take a solution for $I_1, \ldots, I_{k-1}$, or

2) Take a solution for $I_1, \ldots, I_{k'}$ and add job I_k.

- Either way, we will get a feasible solution of value

$$T[k-1] \qquad \text{or} \qquad T[k'] + w_k$$

- We choose the better of the two ways!
- The optimal solution is also obtained in one of these two ways.
  - since it is an optimal solution, it makes the same choice!

Implementation

sort all jobs by their finish time t_i: $t_1 \leq t_2 \leq \cdots \leq t_n$

create table $T[0:n]$

$T[0] = 0$

for $k = 1$ to n do

  let k' be the last job such that $t_{k'} \leq s_k$ (*find using binary search*)

  $T[k] = \max(T[k-1], T[k'] + w_k)$

end

Running time $O(n \log n)$

Typesetting Problem

Text typeset in different programs

(text is from "*Eye of the Hurricane: An Autobiography*" by *Richard E. Bellman*)

Plain

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is: Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So, I used it as an umbrella for my activities.

LaTeX

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is: Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. Im not using the term lightly; Im using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word programming. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So, I used it as an umbrella for my activities.

Word

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is: Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So, I used it as an umbrella for my activities.

Typography

Good	Bad
Good fonts.	Bad fonts.
Proportional or variable width fonts.	Monospaced or fixed-width fonts.
…	…
Good text alignment or justification.	

Full Justification

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is: Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. Im not using the term lightly; Im using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word programming. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So, I used it as an umbrella for my activities.

How to Break a Paragraph into Lines?

Given: A paragraph of text.

Goal: Break the paragraph into lines in the optimal way.

How to Break a Paragraph into Lines?

Given: A paragraph of text.

Goal: Break the paragraph into lines in the optimal way.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut.

bad: spaces between words are too large

underfull: not enough text to fill the line

overfull: there is too much text for one line

How to Break a Paragraph into Lines?

Given: A paragraph of text.

Goal: Break the paragraph into lines in the optimal way.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut.

bad: spaces between words are too large

underfull: not enough text to fill the line

overfull: there is too much text for one line

However, the rules are complex and subjective…

How to Break a Paragraph into Lines?

Given: A paragraph of text $s = s_1 \dots s_n$ that consists of n characters.

Also, a typographer has implemented and given us a function

$$b\colon \text{substring} \mapsto \text{badness score}$$

$\text{badness} = 0 \Rightarrow$ we can typeset the substring on a single line s.t. it looks good

$\text{badness} \gg 100 \Rightarrow$ we cannot typeset the substring on a single line s.t. it looks good

Goal: Choose $k \geq 1$ and split s into k lines s.t. their total badness is minimal.

(in real life, we need to treat the first and last line differently)

How to Break a Paragraph into Lines?

Given: A paragraph of text $s = s_1 \dots s_n$ that consists of n characters.

Goal: Choose $k \geq 1$ and split s into k lines s.t. their total badness is minimal.

s_1	$\cdots$	s_{a_1}
s_{a_1+1}	$\cdots$	s_{a_2}
	$\ddots$	
$s_{a_{k-1}+1}$	$\cdots$	s_n

find breakpoints $a_1, \dots, a_{k-1}$ so as to minimize

$$cost = \sum_{i=1}^{k} b(s_{a_{i-1}+1}, \dots, s_{a_i})$$

assume $a_0 = 0$ and $a_k = n$

Dynamic Program

$$\sum_{i=1}^{k} b(s_{a_{i-1}+1}, \ldots, s_{a_i})$$

s_1	$\cdots$	s_{a_1}
s_{a_1+1}	$\cdots$	s_{a_2+1}
	$\ddots$	
$s_{a_{k-1}+1}$	$\cdots$	s_n

Subproblems?

Typeset the prefix $s_1 \ldots s_i$ of s

DP Table?

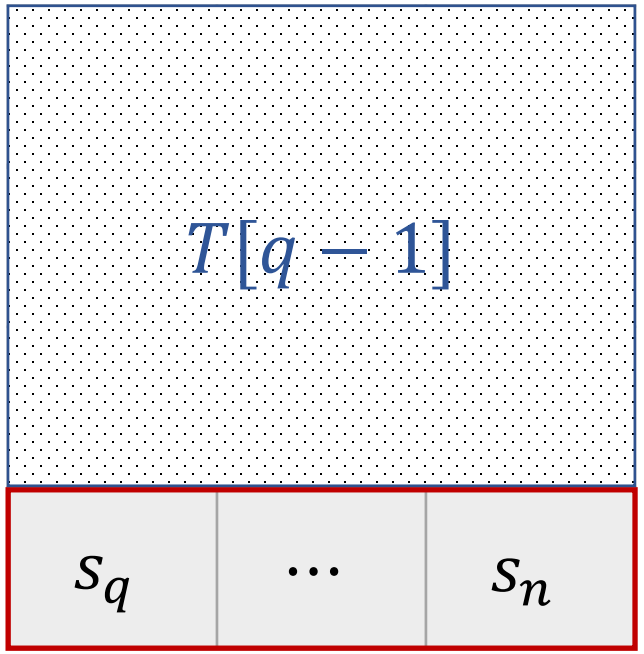
Table $T[0:n]$. Entry $T[i]$ equals the optimal cost of typesetting $s_1 \ldots s_i$.

Initialization?

$T[0] = 0$

DP Recurrence

$$\sum_{i=1}^{k} b(s_{a_{i-1}+1}, \ldots, s_{a_i})$$



We can construct solution by finding an optimal solution for $s_1 \ldots s_{q-1}$ (for some $q \geq 1$) and then typesetting string

$$s_q \ \ldots s_n$$

on a single line. The cost is

$$T[q-1] + b(s_q \ldots s_n).$$

By choosing optimal q, we get a solution of cost

$$\min_{1 \leq q \leq n} T[q-1] + b(s_q \ldots s_n)$$

DP Recurrence

$$\sum_{i=1}^{k} b(s_{a_{i-1}+1}, \ldots, s_{a_i})$$

s_1	$\cdots$	s_{a_1}
s_{a_1+1}	$\cdots$	s_{a_2+1}
	$\ddots$	
$s_{a_{k-1}+1}$	$\cdots$	s_n

We can construct solution by finding an optimal solution for $s_1 \ldots s_{q-1}$ (for some $q \geq 1$) and then typesetting string

$$s_q \ldots s_n$$

on a single line.

The optimal solution also works like this for $q = a_{k-1} + 1$

DP Recurrence

s_1	$\cdots$	s_{a_1}
s_{a_1+1}	$\cdots$	s_{a_2+1}
	$\ddots$	
$s_{a_{k-1}+1}$	$\cdots$	s_i

We obtain recurrence

$$T[i] = \min_{1 \le q \le i} T[q-1] + b(s_q \ldots s_i)$$

The algorithm fills out entries one-by-one:

$$T[1], \ldots, T[n]$$

It needs time $O(i)$ to fill entry $T[i]$.

Running time: $O(n^2)$ assuming that computing $b(s_q \ldots s_i)$ takes constant time.

Knapsack

Knapsack

Given: a set of n items: $1, \ldots, n$. Each item i has value v_i and weight w_i.

Goal: choose a subset of items S of total weight at most W so as to maximize their value.

We will assume that all weights w_i and W are positive integers.

Attempt #1

Subproblem #i

Knapsack with items $1, \ldots, i$.

DP Table

The value of the optimal solution for items $1, \ldots, i$.

Recurrence

Option 1: $i \notin S$ then $T[i] = T[i-1]$

Attempt #1

Option 2: $i \in S$

We would like to take an optimal solution for $\mathcal{P}_j$ and add item i to it.

… but for all we know the optimal solution for $\mathcal{P}_j$ may have weight W

… if we add item i to it, we will get an infeasible solution

DP for Knapsack

Subproblem (i, W')

Knapsack with items $1, \ldots, i$ and capacity W'.

DP Table $T[i, W']$

The value of the optimal solution for items $1, \ldots, i$ and capacity W'.

Recurrence

Option 1: $i \notin S$ then $T[i, W'] = T[i-1, W']$.

Any feasible solution for $(i-1, W')$ is also a feasible solution for (i, W').

DP for Knapsack

Option 2: $i \in S$. Possible only when $w_i \leq W'$.

We can get an optimal solution for $(i-1, W'-w_i)$ and add item i.

We get a solution of weight $(W'-w_i)+w_i = W'$ as required!

It's value

$$T[i-1, W'-w_i] + v_i$$

The optimal solution also chooses option 1 or 2.

$$T[i, W'] = \begin{cases} \max(T[i-1, W'], T[i-1, W'-w_i] + v_i), & \text{if } w_i \leq W' \\ T[i-1, W'], & \text{otherwise} \end{cases}$$

Implementation

Create table $T[0:n, 0:W]$

Initialize $T[0,*] = 0$ (*if there are not items, we cannot choose any*)

for $i = 1$ to n

    for $W' = 0$ to W do

$$T[i, W'] = \begin{cases} \max(T[i-1, W'], T[i-1, W' - w_i] + v_i), & \text{if } w_i \leq W' \\ T[i-1, W'], & \text{otherwise} \end{cases}$$

return $T[n, W]$

Running time: $O(nW)$

Space/Memory: $O(nW)$.