Dynamic Programming

Shuang Zhao

Microsoft Research Asia

September 5, 2005

Outline

Introduction

What is Dynamic Programming?

Definition

Dynamic Programming is a technique for efficiently recurrence computing by storing partial results.

In this slides, I will NOT use too many formal words, but only look on some *interesting problems*.

The First Problem

Longest Ascending Subsequence

- P : $a_1, a_2, \ldots, a_n$.
- Q : $a_{b_1}, a_{b_2}, \ldots, a_{b_k}$, satisfying

$$1 \leq b_1 < b_2 < \ldots < b_k \leq n$$

and

$$a_{b_1} < a_{b_2} < \ldots < a_{b_k}$$

- We say Q is an ascending subsequence of P.
- Your task is: given a sequence P, find the length its longest ascending subsequence (**LAS**).

The First Problem

Longest Ascending Subsequence

- We use f_i to denote the length of P's LAS *ending* with element a_i.
- Let $a_0 := -\infty, f_0 := 0$. Then we have

$$f_k = \max_{0 \leq i < k} \{f_i + 1 : a_i < a_k\}, \quad 1 \leq k \leq n$$

- Hence the length of P's LAS is $\max\{f_i\}$.
- This naive Dynamic Programming algorithm runs in $O(n^2)$ time, and later we will look on this problem again.

The Second Problem

Shortest Hamilton Path

- Given a connected graph $G(V, E)$ and a vertex $s \in V$.
- Given a *weight function* w over E, denoting the lengths of edges.
- Your task is to find the length of shortest Hamilton path starting from s.

The Second Problem

Shortest Hamilton Path

- This problem is $\mathcal{NP}$-hard.
- When $|V|$ is small, one FAST algorithm to solve this problem is Dynamic Programming .
- Use $f_{i,S'}$ where $i \in V, S' \subseteq S$ to denote the length of shortest Hamilton path over S', which ending at vertex i.
- Then $\min_{i \in V} \left\{ f_{i,S} \right\}$ is what we want, and

$$f_{i,S'} = \min_{j \in S'} \left\{ f_{j,S'-\{i\}} + w(j,i) \right\}$$

- This algorithm runs in $O(|V|^2 \cdot 2^{|V|})$ time.

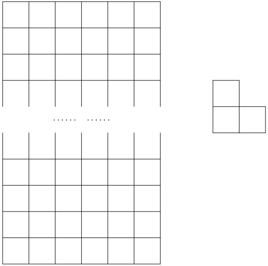
Partial Result

Problem I

Tiling Counting

Tiling Counting

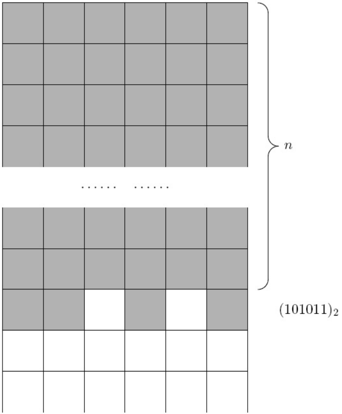
- The Problem: There is a board with N rows and 6 columns. How many ways can we cover the board with 'L' pieces?



- One way to solve this problem, is Dynamic Programming .
- But where are the partial results?

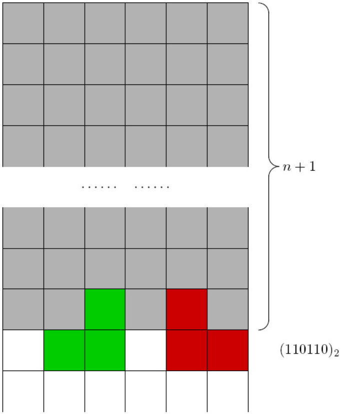
Tiling Counting

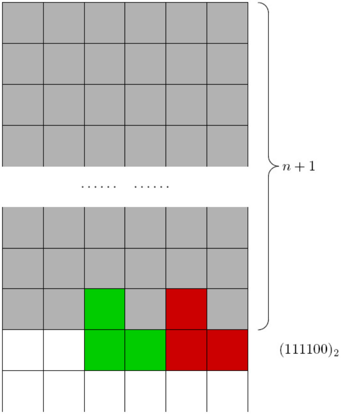
- This is a pattern, say $P_{n,(101011)_2}$:



Tiling Counting

- And by completely covering the last row of $P_{n,(101011)_2}$, we can obtain $P_{n+1,(110110)_2}$ and $P_{n+1,(111100)_2}$.





Tiling Counting

Observation

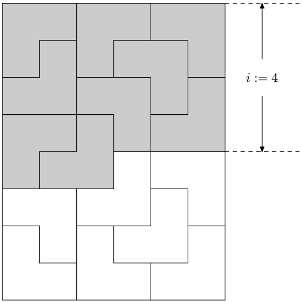
For any pattern $P_{i,j}$, after covering its $(i+1)$-th row, we can only get patterns with the form of $P_{i+1,j'}$.

We call $P_{i+1,j'}$ can be generated from $P_{i,j}$, denoting as $P_{i+1,j'} \succ P_{i,j}$.

Tiling Counting

Observation

For any given covering of the $N \times 6$ board and an integer i $(0 \leq i < N)$, there will be one and only one pattern $P_{i,j}$, which is contained in the covering.



Tiling Counting

The partial results are:

- Let $f_{i,j}$ to be the number of ways to cover pattern $P_{i,j}$.
- And

$$f_{i,j} = \begin{cases} 1 & j = 0 \\ 0 & \text{otherwise} \end{cases} \qquad i = 0$$
$$f_{i,j} = \sum_{P_{i,j} \succ P_{i-1,j'}} f_{i-1,j'} \quad 0 < i < n$$

- Finally $f_{n-1,(111111)_2}$ is the answer to this problem.
- By a coarse calculation, we know this algorithm runs in $O(2^6 \cdot N \cdot 4^6) = O(N)$ time.

String Counting

String Counting

- The Problem: Given an alphabet Σ, a set of strings $S \subseteq_f \Sigma^*$, and an integer n. Your task is to count the number of n-length-strings which contain at least one string in S as its substring.

String Counting

- Let $\Sigma := \{a, b, c\}$, $S := \{ab\}$ and $n := 3$. Then we have:

  a<u>ab</u>   <u>ab</u>a   <u>ab</u>b   <u>ab</u>c   b<u>ab</u>   c<u>ab</u>

- One method to solve this problem is using the *inclusion and exclusion theorem*. But the time complexity of this method is quite high (at least $O(2^{|S|})$).

String Counting

Definition

Let $\trianglelefteq$ and $\trianglerighteq$ be two binary relations over Σ^*:

$$x \trianglelefteq y \iff x \text{ is a prefix of } y$$
$$x \trianglerighteq y \iff x \text{ is a suffix of } y$$

Definition

We define the prefix set of S ($S \neq \emptyset$) by

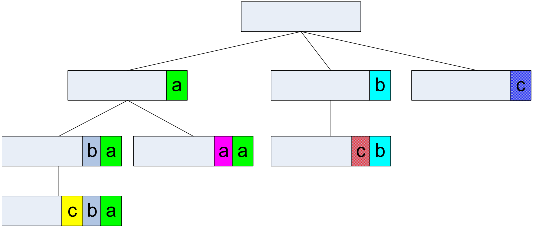
$$\text{pre}(S) := \{s' : (\exists s \in S)(s' \trianglelefteq s)\}$$

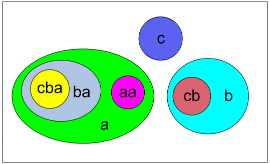
Obviously, $S \subseteq \text{pre}(S)$.

String Counting

For instance, let $\Sigma := \{a, b, c\}$ and $S := \{aa, ba, cba\}$, then

$$\text{pre}(S) = \{\varepsilon, a, aa, b, ba, c, cb, cba\}$$





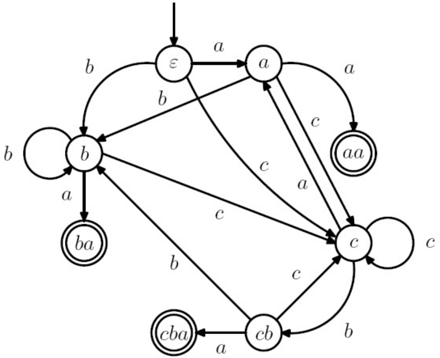
String Counting

- Let $\mathrm{pre}(S) = \{s_0, s_1, \ldots, s_t\}$ where $s_0 = \varepsilon$.
- Let F_i $(i = 0, 1, \ldots, t)$ be the subset of Σ^*, satisfying for all $s' \in F_i$:
  - $s_i \trianglerighteq s'$
  - $\neg(\exists s'' \triangleleft s')[(\exists r \in S)(r \trianglerighteq s'')]$
  - $\neg\exists s_j(s_i \triangleright s_j \wedge s_j \trianglerighteq s')$

String Counting

Observation

Given $p \in \Sigma$, for all $s \in F_i$, there will be exact one j $(0 \leq j \leq t)$ satisfying $s \odot p \in F_j$. We say $F_i \succ F_j$.



String Counting

- Define $f_{i,j}$ by the number of i-length strings in F_j.
- The we have

$$f_{i,j} = \sum_{F_j \succ F_k} f_{i-1,k}$$

and the number of strings which contain at least one string in S as their *substring* is

$$\sum_{i=1}^{n} \sum_{s_j \in S} f_{i,j} \cdot |\Sigma|^{n-i}$$

String Counting

- A brute-force approach to determine the binary relation $\succ$ takes $O(|\text{pre}(S)| \cdot |\Sigma| \cdot L) \approx O(L^2)$ where L is the total length of strings in $\text{pre}(S)$.
- The time complexity of doing the Dynamic Programming is $O(n \cdot |\text{pre}(S)| \cdot |\Sigma|) \approx O(n \cdot |\text{pre}(S)|) \approx O(n \cdot L)$.

Optimization

Approach 1

Speed up partial results' calculation

Speed up partial results' calculation

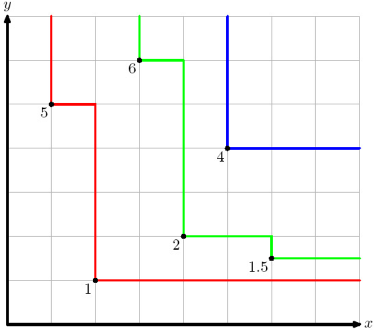
Longest Ascending Subsequence (**LAS**) problem revisited:

- The naive Dynamic Programming algorithm runs in $O(n^2)$ time.
- Can we solve this problem more *efficiently*?

Speed up partial results' calculation

Let $n := 6$, $\{a_n\} := \{5, 1, 6, 2, 4, 1.5\}$:

- *Vertices, Contours*



- Red – 1, Green – 2, Blue – 3, etc.

Speed up partial results' calculation

Proposition

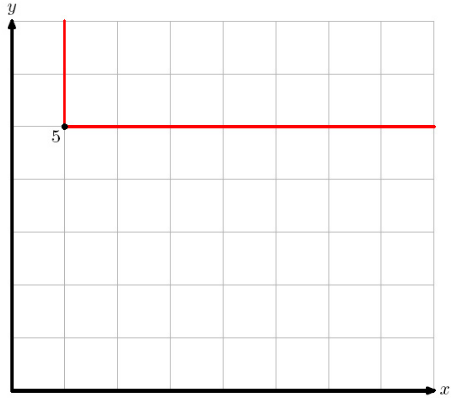
The contours will NOT intersect.

Proof.

Assume two contours with levels i and j ($i < j$) intersects.
Then it holds that at least one vertex of level j is below contours i.
This gives the contradiction that one vertex of level i is below contours i. $\square$

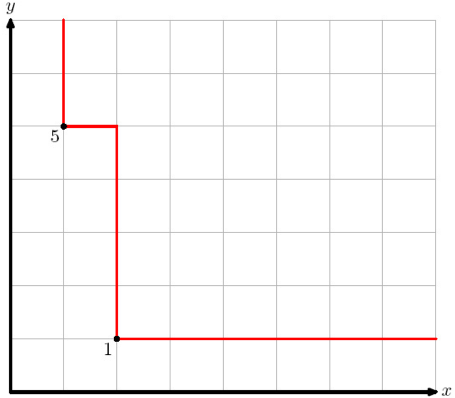
Speed up partial results' calculation

$\{a_n\} = \{5\}$



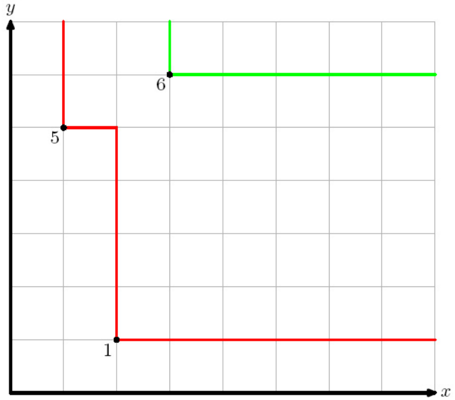
Speed up partial results' calculation

$\{a_n\} = \{5, 1\}$



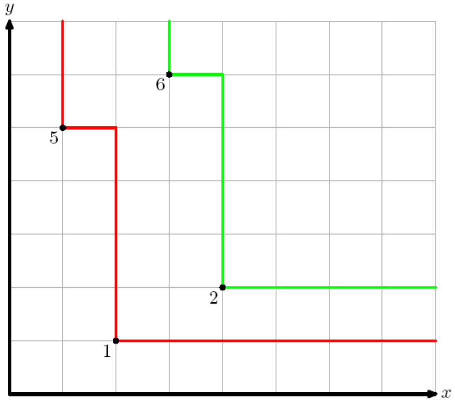
Speed up partial results' calculation

$\{a_n\} = \{5, 1, 6\}$



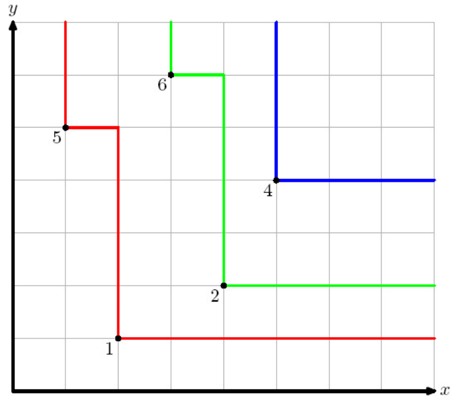
Speed up partial results' calculation

$\{a_n\} = \{5, 1, 6, 2\}$



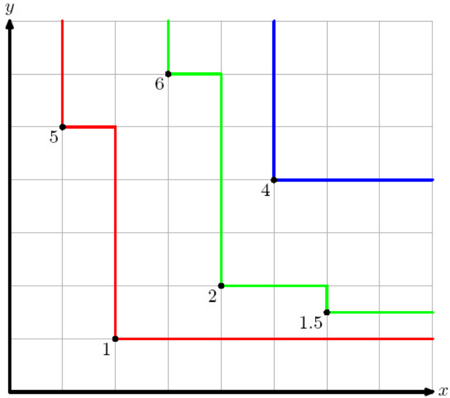
Speed up partial results' calculation

$\{a_n\} = \{5, 1, 6, 2, 4\}$



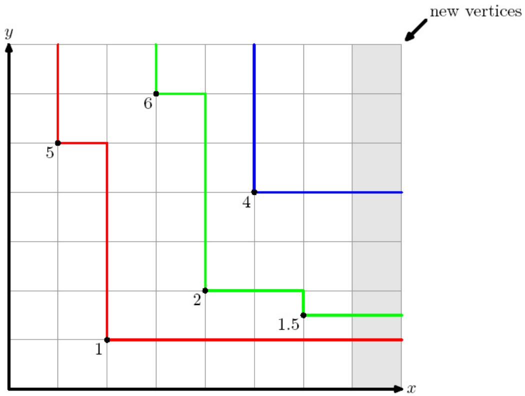
Speed up partial results' calculation

$\{a_n\} = \{5, 1, 6, 2, 4, 1.5\}$



Speed up partial results' calculation

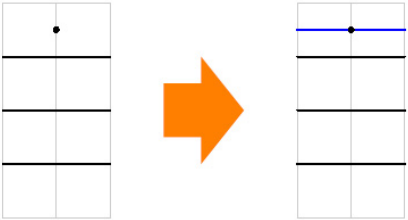
How to update the contours?



Speed up partial results' calculation

How to update the contours?

1. When the new vertex is *above* all contours

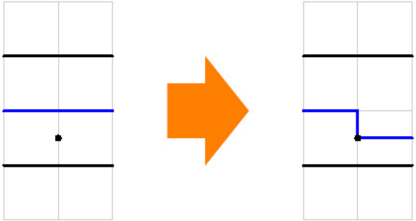


Then a new contour is created.

Speed up partial results' calculation

How to update the contours?

2. When the new vertex is *below* some contour

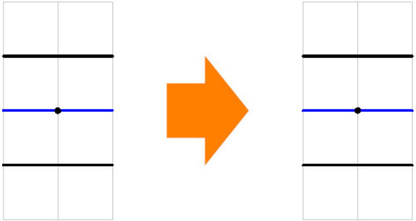


Then the contour *immediately above* the new vertex, is lowered.

Speed up partial results' calculation

How to update the contours?

3. When the new vertex is just on some contour



Then the contours remain the same.

Speed up partial results' calculation

Find the contour immediately above the new vertex

- A brute-force approach runs in $O(n)$ time.
- A Binary Search algorithm takes only $O(\log n)$ time.

So by using Binary Search, the time complexity of entry algorithm is reduced to $O(n \log n)$.

Speed up partial results' calculation

Further thinking

- How to implement this algorithm?
- How to find the longest non-descending subsequence of a given sequence?
- If we given a *weight* to every number in the sequence, how to find the ascending subsequence which maximizes the sum of weights?

Approach 2

Speed up by monotonicity

Speed up by monotonicity

The *Optimal Binary Search Tree* (**OBST**) Problem

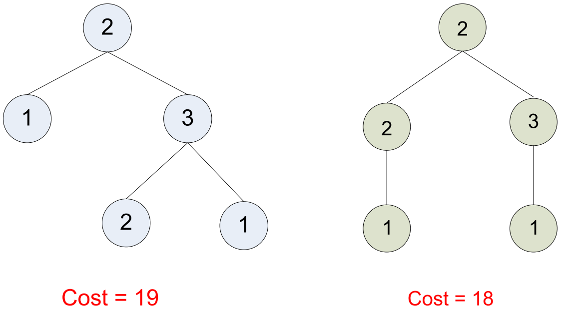
- n numbers $a_1 < a_2 < \ldots < a_n$
- n weights $w_1, w_2, \ldots, w_n \geq 0$
- Construct a binary search tree using $a_1, \ldots, a_n$.

$$Cost = \sum_{i=1}^{n} w_i \cdot d_i$$

- Your task is to find a *binary search tree* which minimizes the cost.

Speed up by monotonicity

- Let $n := 5$ and $\{w_n\} := \{1, 2, 2, 3, 1\}$.



- The cost of OBST is 18.

Speed up by monotonicity

The naive Dynamic Programming approach

- $f_{i,j}$: The cost of OBST constructed by $w_i, w_{i+1}, \ldots, w_j$
- Then

$$f_{i,j} = 0 \quad (i > j)$$

$$f_{i,j} = \min_{i \le k \le j} \{f_{i,k-1} + f_{k+1,j}\} + \sum_{i \le t \le j} w_t \quad (i \le j)$$

- The answer is $f_{1,n}$.
- The time complexity is $O(n^3)$.
- How to speed up this algorithm?

Speed up by monotonicity

Definition

For a given $m \times n$ matrix A, if for all $i_1 \leq i_2 \leq j_1 \leq j_2$, it holds

$$A[i_1, j_1] + A[i_2, j_2] \leq A[i_1, j_2] + A[i_2, j_1]$$

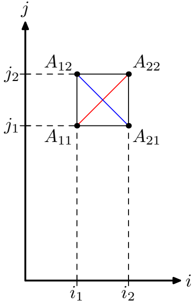
then we say A is totally monotonic.

Speed up by monotonicity

The inequality

$$A[i_1, j_1] + A[i_2, j_2] \leq A[i_1, j_2] + A[i_2, j_1]$$

is called Quadrangle Inequality.



Speed up by monotonicity

Recurrent formula of OBST problem

$$f_{i,j} = 0 \quad (i > j)$$

$$f_{i,j} = \min_{i \le k \le j} \{f_{i,k-1} + f_{k+1,j}\} + \sum_{i \le t \le j} w_t \quad (i \le j)$$

Define $F, W \in \mathbb{R}_+^{n \times n}$ by

$$F[i,j] := f_{i,j} \quad W[i,j] := \sum_{i \le k \le j} w_k$$

Then when $i \le j$

$$F[i,j] = \min_{i \le k \le j} \{F[i,k-1] + F[k+1,j]\} + W[i,j]$$

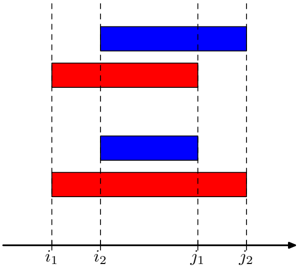
Speed up by monotonicity

Proposition

Matrix W is totally monotonic.

Proof.

For all $i_1 \leq i_2 < j_1 \leq j_2$,
$W[i_1, j_1] + W[i_2, j_2] = W[i_1, j_2] + W[i_2, j_1]$ □



Speed up by monotonicity

Proposition

Matrix F is also totally monotonic.

This can be proved by induction on $j_2 - i_1$. To see the details, read Proof 1 in the Appendix section.

Speed up by monotonicity

Definition

For all $1 \le i \le j \le n$, define $s(i,j)$ by

$$s(i,j) := \max_{i \le k \le j} \left\{ k : F[i,j] = F[i,k-1] + F[k+1,j] + W[i,j] \right\}$$

Speed up by monotonicity

Proposition

$s(i,j)$ is monotonic, namely for all $1 \leq i \leq j < n$,

$$s(i,j) \leq s(i,j+1) \leq s(i+1,j+1)$$

To see the proof, read Proof 2 in the Appendix section.

Speed up by monotonicity

- The new recurrent formula

$$f_{i,j} = \min_{s(i,j-1) \le k \le s(i+1,j)} \{f_{i,k-1} + f_{k+1,j}\} + \sum_{i \le t \le j} w_t \quad (i \le j)$$

- The time complexity to solve it is $O(n^2)$.

Appendix

Proof 1

(In the OBST problem)

Proposition

Matrix F is also totally monotonic.

Proof (Part 1).
When $i_1 = i_2$ or $j_1 = j_2$ the *Quadrangle Inequality* holds. $\square$

Proof 1

Proof (Part 2).
When $i_1 < i_2 = j_1 < j_2$, we prove by induction on $j_2 - i_1$.
Let

$$F[i_1, j_2] = F[i_1, k-1] + F[k+1, j_2] + W[i_1, j_2]$$

Without loss of generality, $k \leq j_1$.

Proof 1

Case 1. $k < j_1 (= i_2)$

$$
\begin{aligned}
& F[i_1, j_1] + F[i_2, j_2] \\
\le\ & F[i_1, k-1] + F[k+1, j_1] + W[i_1, j_1] + F[i_2, j_2] \\
\le\ & F[i_1, k-1] + W[i_1, j_1] + F[k+1, j_2] + F[i_2, j_1] \\
\le\ & F[i_1, k-1] + W[i_1, j_2] + F[k+1, j_2] + F[i_2, j_1] \\
=\ & F[i_1, j_2] + F[i_2, j_1]
\end{aligned}
$$

Proof 1

Case 2. $k = j_1 (= i_2)$

$$
\begin{aligned}
& F[i_1, j_1] + F[i_2, j_2] \\
\leq\ & F[i_1, k-1] + W[i_1, j_1] + F[i_2, j_2] \\
\leq\ & F[i_1, k-1] + W[i_1, j_1] + F[j_1 + 1, j_2] + W[i_2, j_2] \\
=\ & F[i_1, k-1] + W[i_1, j_2] + F[k + 1, j_2] + W[i_2, j_1] \\
=\ & F[i_1, j_2] + F[i_2, j_1]
\end{aligned}
$$

$\square$

Proof 1

Proof (Part 3).
When $i_1 < i_2 < j_1 < j_2$, we prove by induction on $j_2 - i_1$. Let

$$F[i_2, j_1] = F[i_2, k-1] + F[k+1, j_1] + W[i_2, j_1]$$
$$F[i_1, j_2] = F[i_1, t-1] + F[t+1, j_2] + W[i_1, j_2]$$

Without loss of generality, $t \leq k$. Hence $i_1 \leq t \leq k \leq j_1$.

Proof 1

Then we have

$$\begin{aligned}
& F[i_1, j_1] + F[i_2, j_2] \\
\le\ & F[i_1, t-1] + F[t+1, j_1] + W[i_1, j_1] \\
& \quad + F[i_2, k-1] + F[k+1, j_2] + W[i_2, j_2] \\
\le\ & F[i_1, t-1] + F[t+1, j_2] + W[i_1, j_2] \\
& \quad + F[i_2, k-1] + F[k+1, j_1] + W[i_2, j_1] \\
=\ & F[i_1, j_2] + F[i_2, j_1]
\end{aligned}$$

$\square$

Proof 2

(In the OBST problem)

Proposition

$s(i,j)$ *is monotonic, namely for all* $1 \le i \le j < n$,

$$s(i,j) \le s(i,j+1) \le s(i+1,j+1)$$

Proof.

By symmetry, we need only to prove that $s(i,j) \le s(i,j+1)$.
When $i = j$, $s(i,j) = i \le s(i,j+1)$.

Proof 2

Next we assume $i < j$. For convenience, we use symbol $F_k[i,j]$ to be the short from of $F[i,k-1] + F[k+1,j] + W[i,j]$. So $F_{s(i,j)}[i,j] = F[i,j]$.

Since matrix F is *totally monotonic*, for all $k \le k' \le j$,

$$F[k+1,j] + F[k'+1,j+1] \le F[k'+1,j] + F[k+1,j+1]$$

Proof 2

Thus

$$\begin{aligned} & F[i,k-1]+F[k+1,j]+W[i,j] \\ & \qquad +F[i,k'-1]+F[k'+1,j+1]+W[i,j+1] \\ \leq\ & F[i,k-1]+F[k+1,j+1]+W[i,j+1] \\ & \qquad +F[i,k'-1]+F[k'+1,j]+W[i,j] \end{aligned}$$

namely

$$F_k[i,j]+F_{k'}[i,j+1]\leq F_k[i,j+1]+F_{k'}[i,j]$$

that is

$$F_k[i,j]-F_{k'}[i,j]\leq F_k[i,j+1]-F_{k'}[i,j+1]$$

Proof 2

Therefore

$$F_{k'}[i,j] \leq F_k[i,j] \to F_{k'}[i,j+1] \leq F_k[i,j+1]$$

For all $k < s(i,j)$, $F_{s(i,j)}[i,j] = F[i,j] \leq F_k[i,j]$.

So $F_{s(i,j)}(i,j+1) \leq F_k(i,j+1)$.

Hence $F_{s(i,j)}[i,j+1] \leq F_k[i,j+1]$.

This gives $s(i,j) \leq s(i,j+1)$. $\square$