Imperial College
London

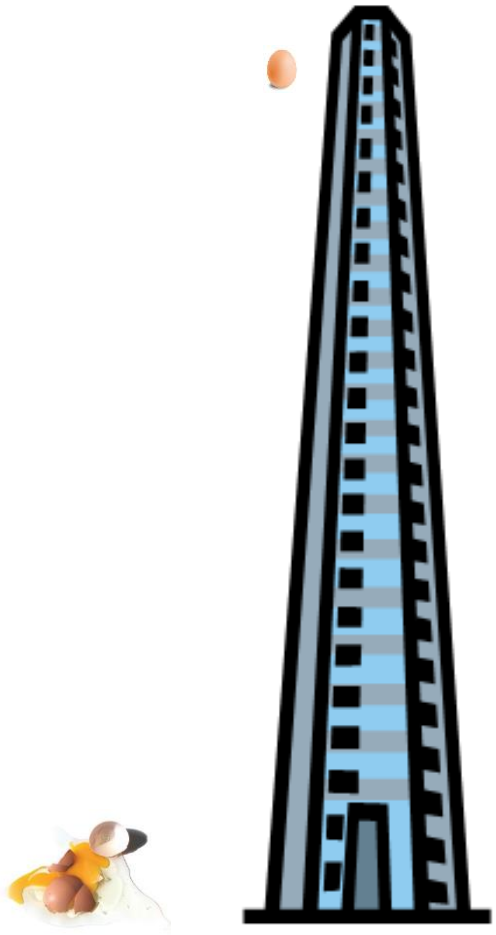
CO202 – Software Engineering – Algorithms

Advanced Algorithms & Puzzles

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

Egg Dropping

Suppose that we wish to know which floors in a K-floor building are safe to drop eggs from, and which floors will cause the eggs to break on landing.



Egg Dropping

Suppose that we wish to know which floors in a K-floor building are safe to drop eggs from, and which floors will cause the eggs to break on landing.

Assumptions

- an egg that survives a fall can be used again, a broken egg must be discarded
- the effect of a fall is the same for all eggs
- if an egg breaks, then it would break if dropped from a higher floor
- if an egg survives a fall, then it would survive a shorter fall
- it is not ruled out that an egg breaks when dropped from the first floor, nor is it ruled out that eggs can survive falls from the K-th floor

Egg Dropping

First question to ask:

How many eggs do we have?

Say, you are given 1 egg. What's your approach?

What if you are given 2 eggs?

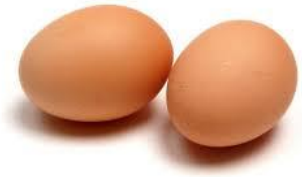
What if you are given N eggs?

What is the optimal egg-dropping policy?

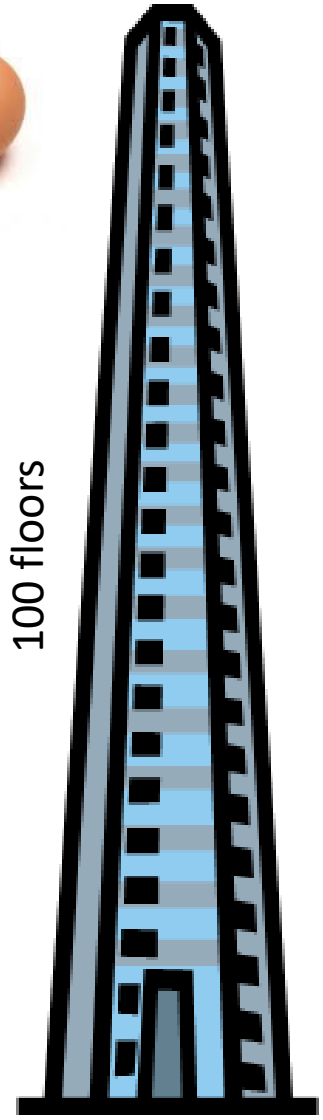
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Egg Dropping

The two egg problem, one hundred floors



- How about binary search?
  - Will take 50 drops worst case.
- What happens if we use the first egg, and go up 10 floors each time it doesn't break?
  - Once the egg is broken, the problem reduces to 1 egg, 9 floors.
  - So, the worst case would need only 19 drops, if critical floor is 99.



Observations

- If the egg broke on floor 10, we only need 9 more drops, so 10 in total
- If the egg broke on floor 11, we need 10 more drops, so 11 in total
- What about jumping 12 floors?
  - We check 12, 24, 48, 60, 72, 84, 96...if it breaks there, still up to 19 drops in total
- The strategy works well at lower floors, but not well on higher floors

[http://datagenetics.com/blog/july22012/index.html]

Egg Dropping

Minimization of maximum regret

We need a strategy that makes things more uniform, that is, we get similar number of egg drops from lower to higher floors

- Imagine we drop our first egg from floor x, if it breaks, then we step through the previous $x - 1$ floors one-by-one.
- If it doesn't break, rather than jumping up another x floors, instead we should step up just $x - 1$ floors (because we have one less drop available if we have to switch to one-by-one).
- So, after checking floor x, the next floor to check is $x + (x - 1)$
- Keep reducing the step by one each time, gives

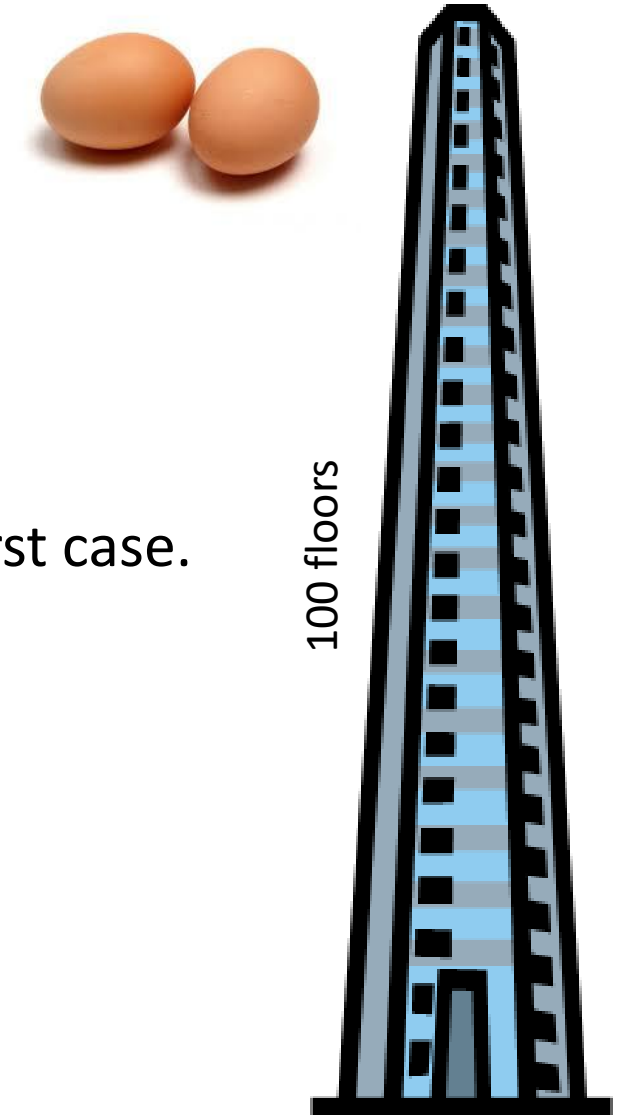
$$x + (x-1) + (x-2) + (x-3) + (x-4) + \cdots + 1 \geq 100$$

$$x(x+1)/2 \geq 100 \qquad x \approx 14$$

Egg Dropping

Drop	Floor
#1	14
#2	17
#3	39
#4	50
#5	60
#6	69
#7	77
#8	84
#9	90
#10	95
#11	99
#12	100

That’s 14 drops in worst case.



Egg Dropping

Let state $s = (n, k)$ be a pair where

- n is the number of test eggs available, $n = 0,1,2, \ldots, N$
- k is the number of consecutive floors yet to be tested, $k = 0,1,2, \ldots, K$

For example, $s = (2,6)$ indicates that two test eggs are available and six consecutive floors are yet to be tested.

The initial state is $s = (N, K)$, and the process terminates either when there are no more test eggs $(n = 0)$ or when all floors have been tested $(k = 0)$, whichever occurs first.

Termination at state $s = (0, k)$ and $k > 0$ means the test failed.

Egg Dropping

Let $w(n, k)$ be the minimum number of trials required to identify the value of the critical floor, given state $s = (n, k)$

- We know, $w(n, 0) = 0$ for all $n > 0$.
- We also know, $w(1, k) = k$ for all k.

If we drop an egg from a floor x, there can be two cases (1) the egg breaks, (2) the egg does not break.

(1) If the egg breaks, we only need to check for floors lower than x with remaining eggs; so the problem reduces to $x - 1$ and $n - 1$

(2) If the egg doesn't break, we only need to check for floors higher than x; so the problem reduces to $k - x$ floors and n eggs

Egg Dropping

If we drop an egg from a floor x, there can be two cases (1) the egg breaks, (2) the egg does not break.

(1) If the egg breaks, we only need to check for floors lower than x with remaining eggs; so the problem reduces to $x - 1$ and $n - 1$

(2) If the egg doesn't break, we only need to check for floors higher than x; so the problem reduces to $k - x$ floors and n eggs

- We want to minimize the number of trials in *worst* case, we take the maximum of the two cases.
- We consider the maximum of above two cases for *every* floor and choose the floor which yields minimum number of trials.

As discussed by Moshe Sniedovich*:

$$w(n,k) = 1 + \min_{x=1,\ldots,k} \{\max(w(n-1,x-1), w(n,k-x))\}$$

*Sniedovich, M. (2003). The joy of egg-dropping in Braunschweig and Hong Kong. INFORMS Transactions on Education, 4(1) 48–64.

Egg Dropping

```
import sys
import numpy as np

def egg_drop(N, K):
    w = np.zeros((N+1, K+1), dtype=np.int64)
    w[1, :] = np.arange(K+1)
    for n in range(2, N+1):
        for k in range(1, K+1):
            w[n, k] = sys.maxsize
            for x in range(1, k+1):
                res = 1 + max(w[n-1, x-1], w[n, k-x])
                if res < w[n, k]:
                    w[n, k] = res
    return w[-1, -1]
```

Task Scheduling

Given a set of n tasks and each task $1 \leq i \leq n$ has an associated (integer-valued) finishing deadline d_i.

Only if a task is executed and completed before (or exactly on) its deadline, a reward r_i is received.

Tasks can be executed one at a time, starting at time point zero and the execution of each task takes exactly one time unit.

Example: $n = 4$ with deadlines $d_1 = 2, d_2 = 1, d_3 = 2, d_4 = 1$ and rewards $r_1 = 100, r_2 = 10, r_3 = 15, r_4 = 27$.

Executing tasks $\{2, 1\}$ yields a reward of 110, while executing tasks $\{4, 1\}$ yields the maximum reward of 127.

We want to find the subset of tasks that maximize our profit.

Task Scheduling

Devise a greedy algorithm `task-selection(d,r)` that solves the task scheduling problem.

Thoughts

- If a task is selected, when should it be executed?
- Which task should we select first?

Task Scheduling

Devise a greedy algorithm `task-selection(d,r)` that solves the task scheduling problem.

Assume the tasks are sorted in descending order of rewards $r_1 \geq r_2 \geq \cdots \geq r_n$.

```
TASK-SELECTION(d,r)
 1: n = d.length
 2: d_max = max(d)
 3: let slots[1..d_max] be a new array
 4: for i=1 to d_max
 5:     slots[i] = 0
 6: reward = 0
 7: for i=1 to n
 8:     for j=d[i] downto 1
 9:         if slots[j] == 0
10:             slots[j] = i
11:             reward = reward + r[i]
12:             break
13: return slots and reward
```

Maximum Subarray Problem

Given a sequence of n real numbers $\{a_1, \dots, a_n\}$, determine a contiguous subsequence $\{a_i, \dots, a_j\}$ for which the sum of its elements $\sum_{k=i}^{j} a_k$ is maximised over all possible subsequences.

Task: Write a recurrence equation that solves subproblems from which the maximum sum can be determined.

Example: $A = \{-2, 1, -3, 4, -1, 2, 1, -5, 4\}$

1	2	3	4	5	6	7	8	9
-2	1	-3	4	-1	2	1	-5	4

Maximum Subarray Problem

Given a sequence of n real numbers $\{a_1, \dots, a_n\}$, determine a contiguous subsequence $\{a_i, \dots, a_j\}$ for which the sum of its elements $\sum_{k=i}^{j} a_k$ is maximised over all possible subsequences.

Task: Write a recurrence equation that solves subproblems from which the maximum sum can be determined.

Example: $A = \{-2, 1, -3, 4, -1, 2, 1, -5, 4\}$

1	2	3	4	5	6	7	8	9
-2	1	-3	4	-1	2	1	-5	4

$$\sum_{k=4}^{7} a_k = 6$$

Maximum Subarray Problem

Let s_j be the maximum sum over all subsequences ending at j. Then the following recurrence determines the solution by considering two cases:

1) extend the maximum sum at position $j-1$ by adding a_j

2) start a new subsequence at index j

$$s_j = \max(s_{j-1} + a_j, a_j)$$

Maximum Subarray Problem

```
def max_subarray(A):
    max_ending_here = max_so_far = A[0]
    for x in A[1:]:
        max_ending_here = max(x, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

A* Search

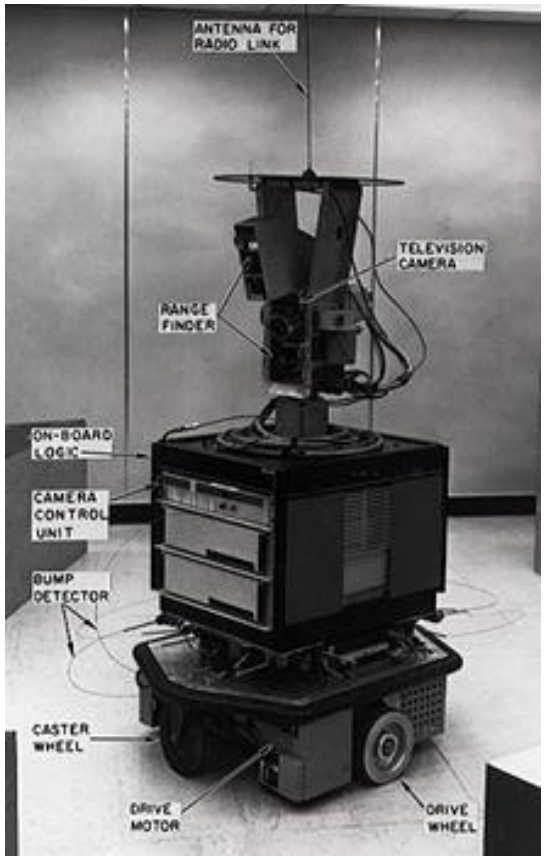
A search algorithm widely used in pathfinding and graph traversal.

Described in 1968 by Peter Hart, Nils Nilsson
and Bertram Raphael of Stanford Research Institute.

It's an extension of Dijkstra which makes use of heuristics to guide its search and often achieves better performance.

A* is an *informed* search algorithm, or best-first search. It solves problems by searching among all possible paths to a goal and first considers the ones that appear best.

Overview
A* constructs a tree of paths from a start node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.



Shakey the Robot

[Wikipedia]

Shakey the Robot

https://youtu.be/7bsEN8mwUB8

A* Search

At each iteration of its main loop, A* determines which of its partial paths to expand based on an estimate of the cost.

Specifically, A* selects the path that minimises

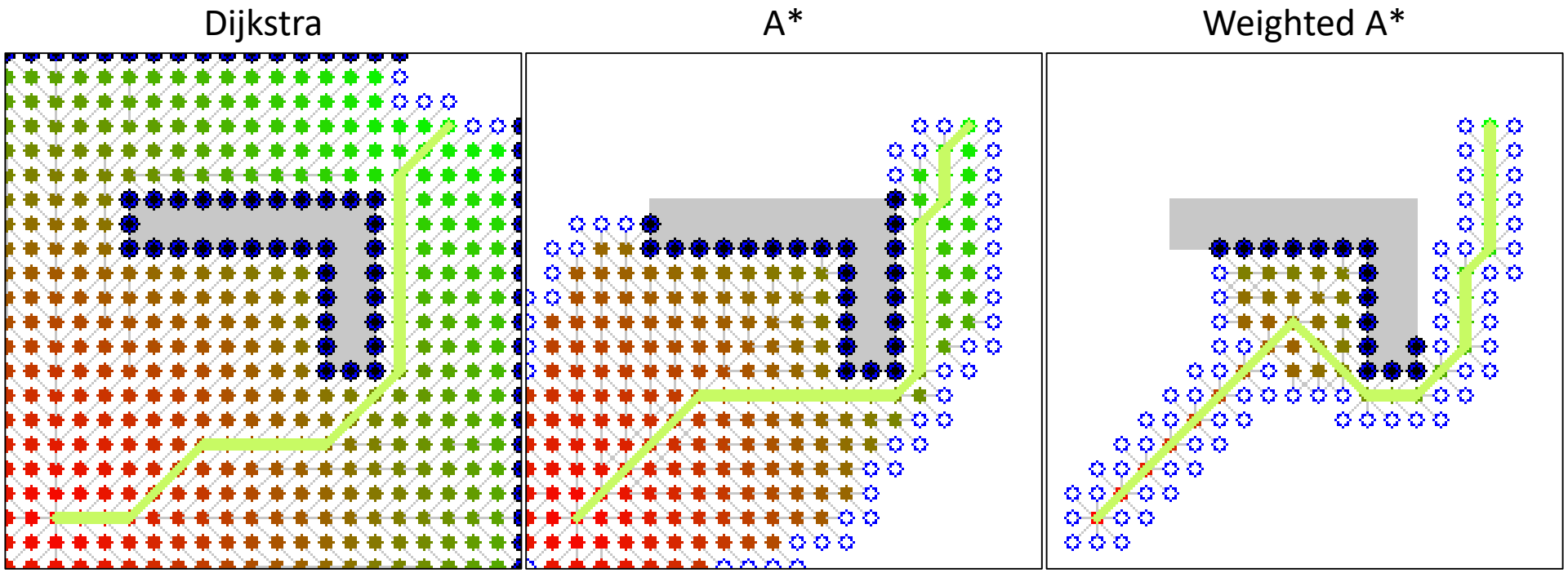
$$f(n) = g(n) + h(n)$$

where n is the last node on the current path, $g(n)$ is the cost of the path from the start to n, and $h(n)$ is a heuristic that estimates the cost of the shortest/cheapest path from n to the goal.

The heuristic function h is problem-specific, and for A* to find the actual shortest path, the heuristic h must be *admissible**.

*Meaning that it never overestimates the actual cost.

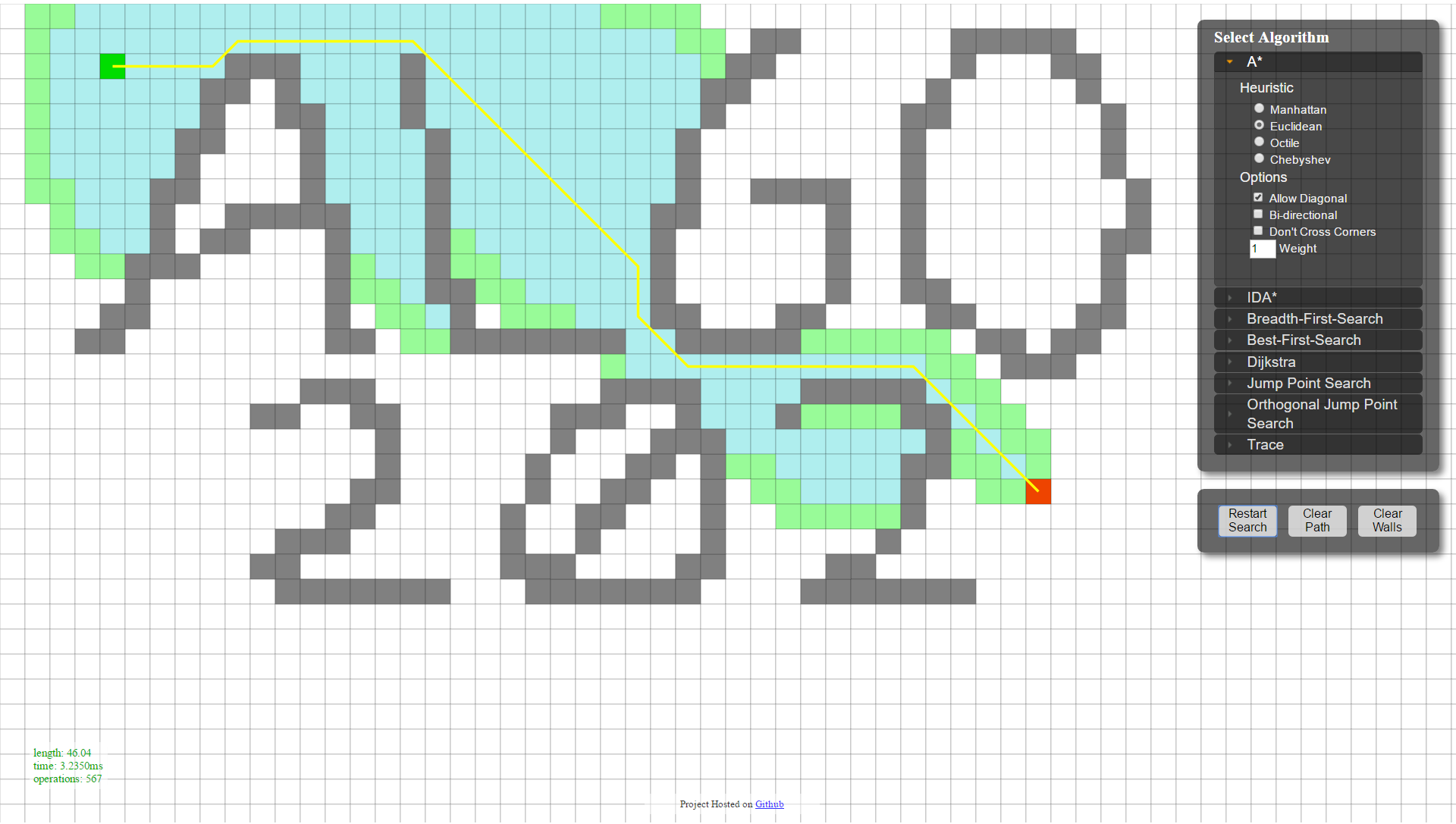
Dijkstra vs A*



Path Finding Competition

https://qiao.github.io/PathFinding.js/visual/

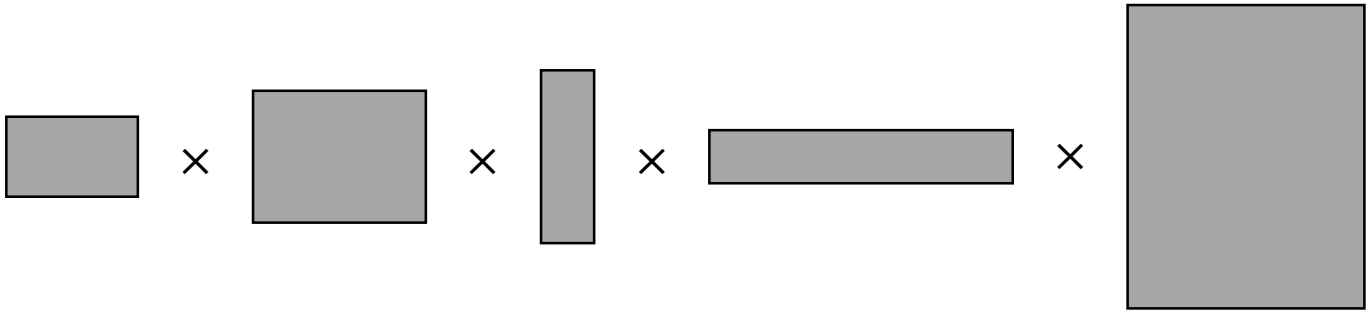
Path Finding Competition



https://qiao.github.io/PathFinding.js/visual/

Matrix Chain Multiplication

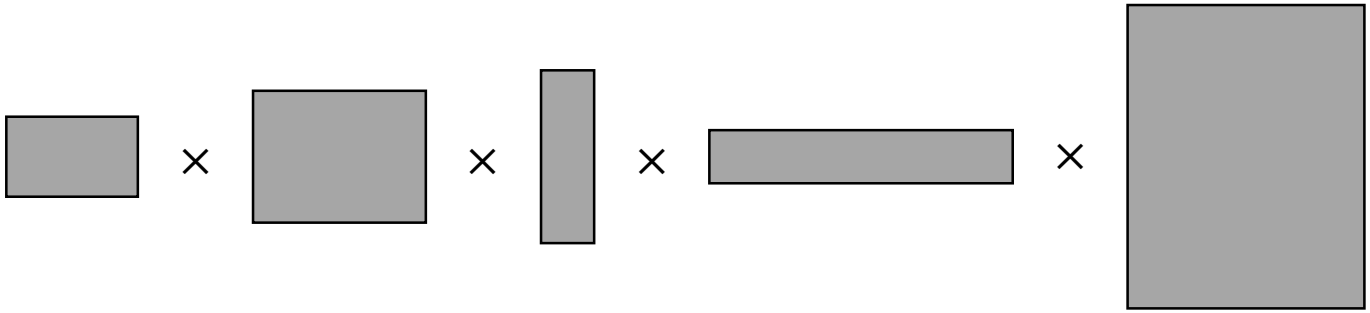
Imagine we have to multiply a chain of matrices of different sizes



$$A_1 \times A_2 \times A_3 \times \cdots A_n$$

Matrix Chain Multiplication

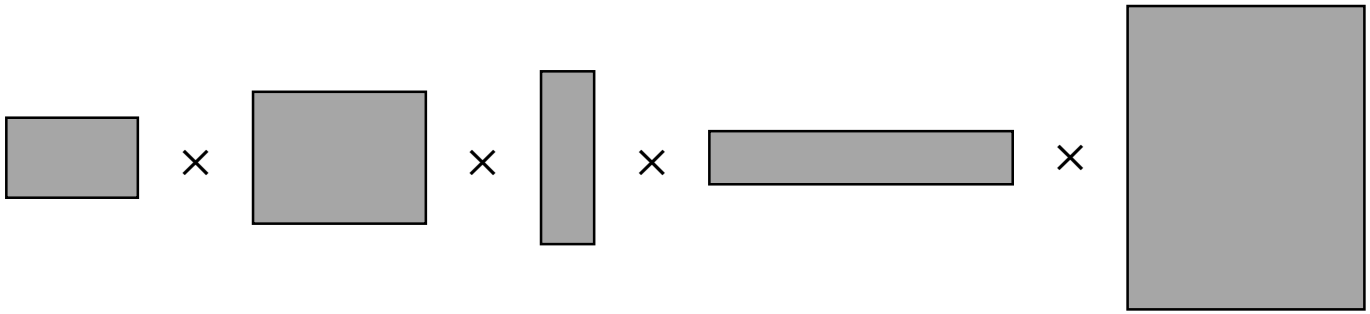
Imagine we have to multiply a chain of matrices of different sizes



$$A_1 \times((A_2 \times A_3)\times \cdots A_n)$$

Matrix Chain Multiplication

Imagine we have to multiply a chain of matrices of different sizes



$$A_1 \times ((A_2 \times A_3) \times \cdots A_n)$$

$$(A_1 \times (A_2 \times A_3)) \times \cdots A_n$$

$$(A_1 \times A_2) \times (A_3 \times \cdots A_n)$$

Matrix Chain Multiplication

Let us consider the following example with three matrices

$$A^{m\times n} \times B^{n\times p} \times C^{p\times s}$$

Case 1. $A \times (B \times C)$

This order will require $nps + mns$ scalar multiplications

Case 2. $(A \times B) \times C$

This order will require $mnp + mps$ scalar multiplications

Say $m = 10$, $n = 100$, $p = 10$ and $s = 1000$, then case 1 requires $2{,}000{,}000$ calculations, while case 2 requires only $110{,}000$.

Matrix Chain Multiplication

Let $m[i,j]$ be the minimum number of scalar multiplications needed to multiply a chain of matrices from matrix i to matrix j (with $i \leq j$)

We split the chain at some matrix k, such that $i \leq k < j$, and determine which combination produces minimum $m[i,j]$.

$$m[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{k=i \ldots j-1} \left(m[i,k] + m[k+1,j] + p_{ikj} \right), & \text{otherwise} \end{cases}$$

$$\left(A_i^{r_i \times c_i} \times \cdots \times A_k^{r_k \times c_k} \right) \times \cdots \times A_j^{r_j \times c_j}$$

Matrix Chain Multiplication

Let $m[i,j]$ be the minimum number of scalar multiplications needed to multiply a chain of matrices from matrix i to matrix j (with $i \leq j$)

We split the chain at some matrix k, such that $i \leq k < j$, and determine which combination produces minimum $m[i,j]$.

$$m[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{k=i \ldots j-1} \left(m[i,k] + m[k+1,j] + p_{ikj} \right), & \text{otherwise} \end{cases}$$

$$\underbrace{\left(A_i^{r_i \times c_i} \times \cdots \times A_k^{r_k \times c_k}\right)}_{A_{ik}^{r_i \times c_k}} \times \underbrace{\cdots \times A_j^{r_j \times c_j}}_{A_{(k+1)j}^{c_k \times c_j}}$$

$$p_{ikj} = r_i \cdot c_k \cdot c_j$$

rows of A_i

columns of A_k

columns of A_j

Matrix Chain Multiplication

```
def matrix_chain(chain):
    n = len(chain)
    m = np.zeros((n, n))
    s = np.zeros((n, n))
    for l in range(1,n):
        for i in range(0,n-l):
            j = i + l
            m[i,j] = float("inf")
            for k in range(i,j):
                p = chain[i].shape[0] * chain[k].shape[1] * chain[j].shape[1]
                q = m[i,k] + m[k+1,j] + p
                if q < m[i,j]:
                    m[i,j] = q
                    s[i,j] = k
    return m, s
```

Matrix Chain Multiplication

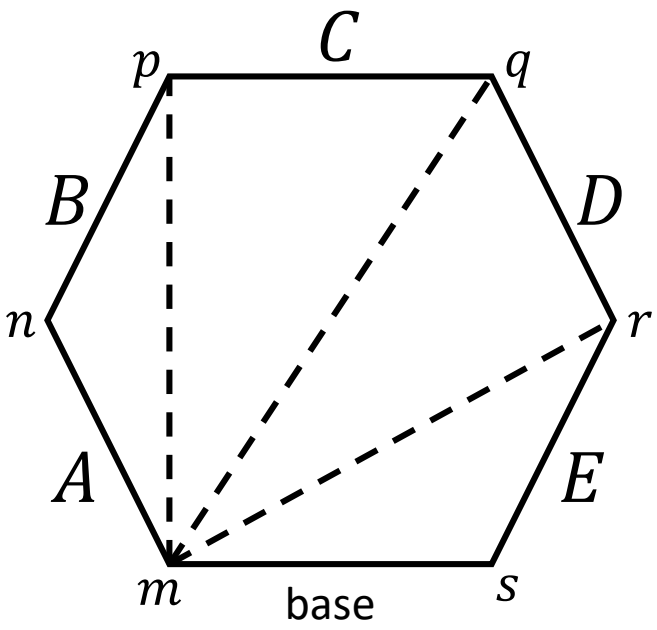
```
def multiply_chain(chain):
    m,s = matrix_chain(chain)
    return matmul_chain(chain,s,0,len(chain)-1)

def matmul_chain(chain,s,i,j):
    if i < j:
        l = matmul_chain(chain,s,i,s[i,j])
        r = matmul_chain(chain,s,s[i,j]+1,j)
        return np.matmul(l,r)
    elif i == j:
        return chain[i]
```

Matrix Chain via Triangulation

A more efficient algorithm for determining the order of multiplications was developed by Hu & Shing in 1981.

They showed that the matrix chain multiplication problem can be transformed into the problem of triangulation of a regular polygon.

Example: $A^{m\times n} \times B^{n\times p} \times C^{p\times q} \times D^{q\times r} \times E^{r\times s}$



The number of different ways of placing parentheses is equal to the number of possible triangulations.

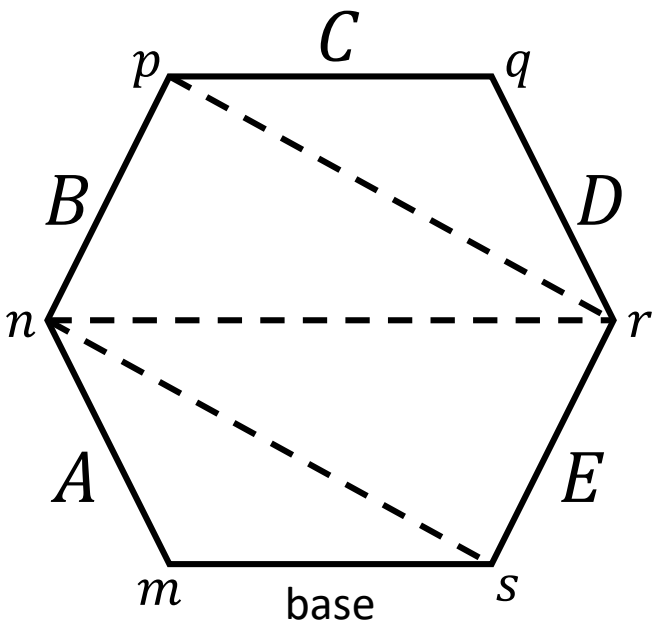
$$\Big(\big((A \times B) \times C\big) \times D\Big) \times E$$

Matrix Chain via Triangulation

A more efficient algorithm for determining the order of multiplications was developed by Hu & Shing in 1981.

They showed that the matrix chain multiplication problem can be transformed into the problem of triangulation of a regular polygon.

Example: $A^{m\times n} \times B^{n\times p} \times C^{p\times q} \times D^{q\times r} \times E^{r\times s}$



The number of different ways of placing parentheses is equal to the number of possible triangulations.

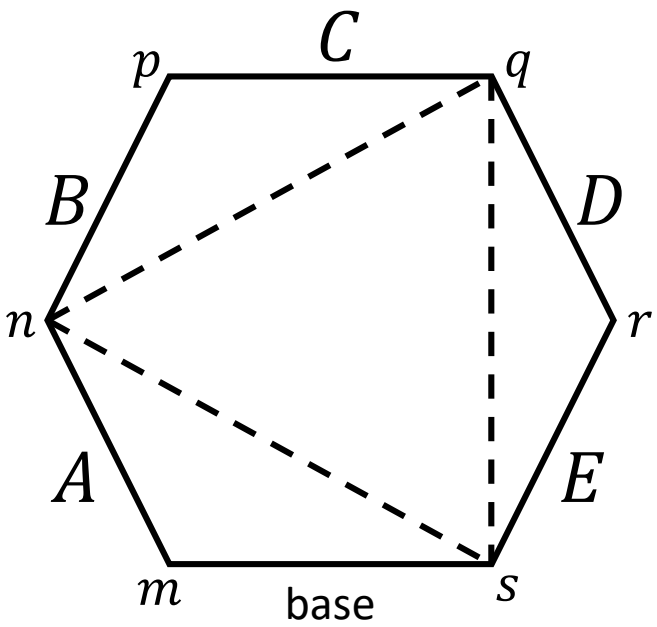
$$A \times \Big(\big(B \times (C \times D) \big) \times E \Big)$$

Matrix Chain via Triangulation

A more efficient algorithm for determining the order of multiplications was developed by Hu & Shing in 1981.

They showed that the matrix chain multiplication problem can be transformed into the problem of triangulation of a regular polygon.

Example: $A^{m\times n} \times B^{n\times p} \times C^{p\times q} \times D^{q\times r} \times E^{r\times s}$



The number of different ways of placing parentheses is equal to the number of possible triangulations.

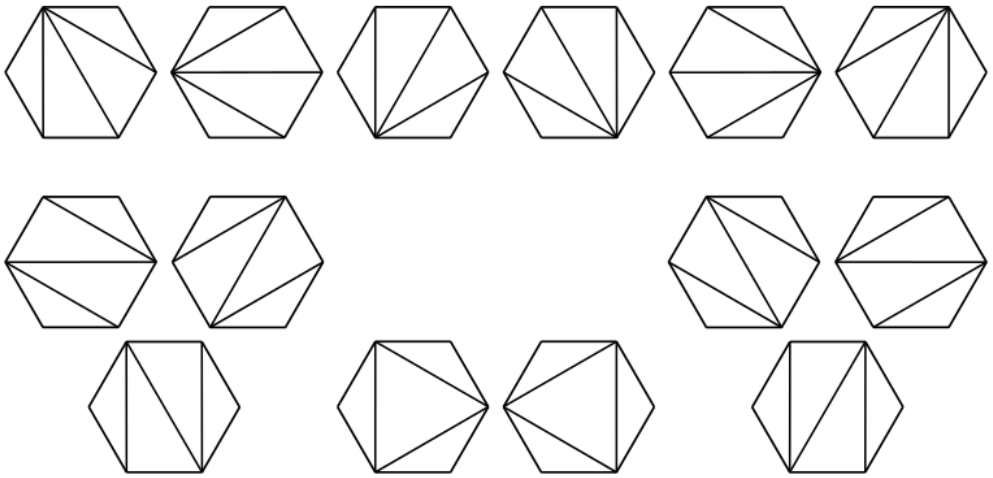
$$A \times \big((B \times C) \times (D \times E)\big)$$

Matrix Chain via Triangulation

A more efficient algorithm for determining the order of multiplications was developed by Hu & Shing in 1981.

They showed that the matrix chain multiplication problem can be transformed into the problem of triangulation of a regular polygon.

Example: $A^{m\times n} \times B^{n\times p} \times C^{p\times q} \times D^{q\times r} \times E^{r\times s}$



For a polygon with $n+1$ sides, there are C_{n-1} possible triangulations.

Catalan number

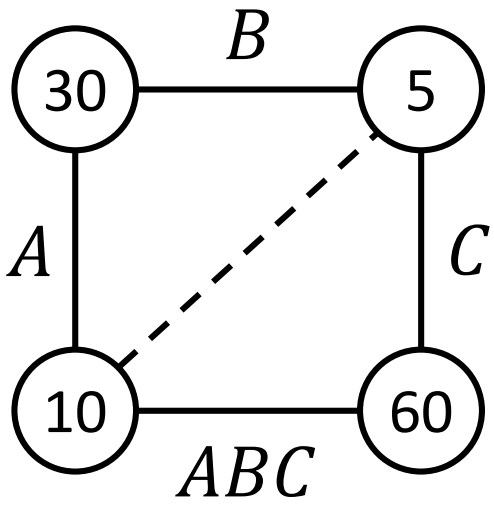
$$C_n = \frac{(2n)!}{(n+1)!\,n!} \quad \text{for } n \geq 0$$

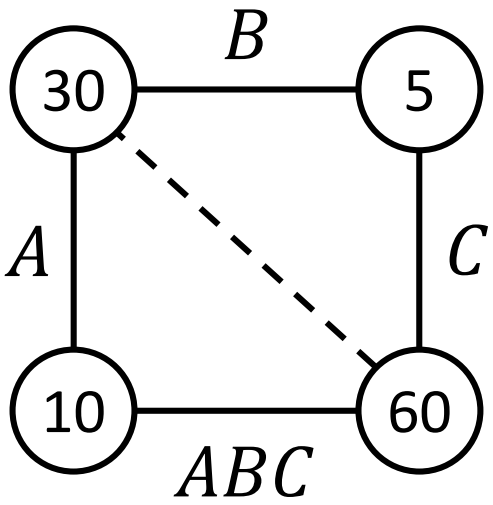
1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Matrix Chain via Triangulation

Back to simple example with three matrices

$$A^{10\times30} \times B^{30\times5} \times C^{5\times60}$$





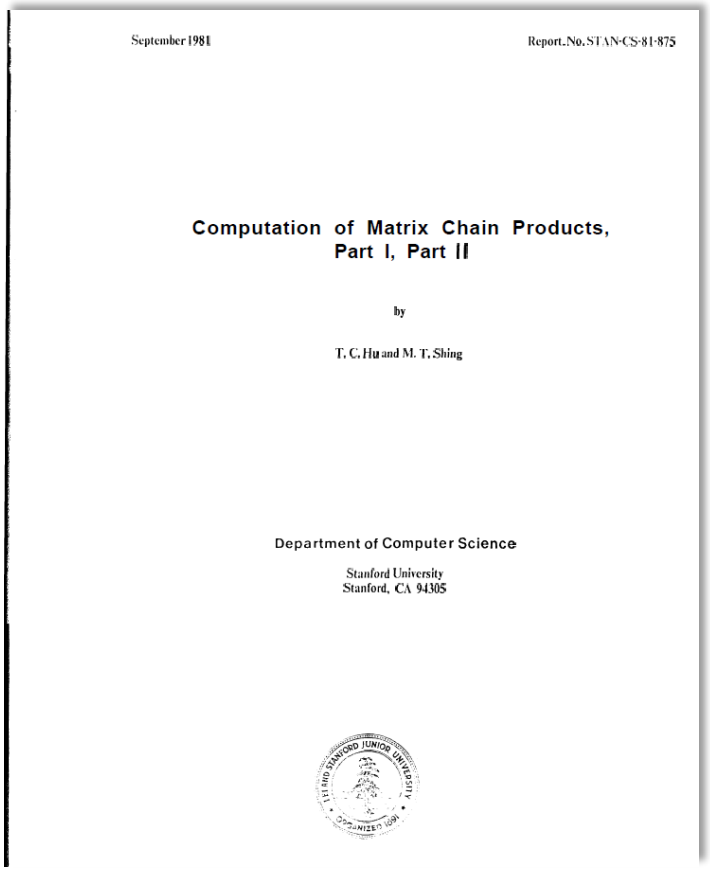
Case 1: $(A \times B) \times C$: $(10 \cdot 30 \cdot 5) + (10 \cdot 5 \cdot 60) = 4500$

Case 2: $A \times (B \times C)$: $(10 \cdot 30 \cdot 60) + (30 \cdot 5 \cdot 60) = 27000$

Matrix Chain via Triangulation

Hu & Shing developed an algorithm that solves the optimal partitioning problem of polygons in $O(n \log n)$.

Their original paper is here.



September 1981 Report No. STAN-CS-81-875

Computation of Matrix Chain Products, Part I, Part II

by

T. C. Hu and M. T. Shing

Department of Computer Science

Stanford University
Stanford, CA 94305

Matrix Multiplication

Consider the multiplication of two square matrices $C = A \times B$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

```
SQUARE-MATMUL(A,B)
 1: n = A.rows
 2: let C be a new n-by-n matrix
 3: for i = 1 to n
 4:     for j = 1 to n
 5:         C[i,j] = 0
 6:         for k = 1 to n
 7:             C[i,j] = C[i,j] + A[i,k] * B[k,j]
 8: return C
```

Running time
$O(n^3)$

Matrix Multiplication with Divide & Conquer

To keep things simple, assume n is exact power of two*

*This is not a requirement, and there are easy ways around that.

Let's partition each of A, B and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Matrix Multiplication with Divide & Conquer

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Each of these equations specifies two multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and the addition of their $\frac{n}{2} \times \frac{n}{2}$ products.

Matrix Multiplication with Divide & Conquer

This yields the following D&C algorithm

```
DC-MATMUL(A,B)
 1: n = A.rows
 2: let C be a new n-by-n matrix
 3: if n == 1
 4:     C[1,1] = A[1,1] * B[1,1]
 5: else partition A, B and C in n/2-by-n/2 matrices
 6:     C_11 = DC-MATMUL(A_11,B_11) + DC-MATMUL(A_12,B_21)
 7:     C_12 = DC-MATMUL(A_11,B_12) + DC-MATMUL(A_12,B_22)
 8:     C_21 = DC-MATMUL(A_21,B_11) + DC-MATMUL(A_22,B_21)
 9:     C_22 = DC-MATMUL(A_21,B_12) + DC-MATMUL(A_22,B_22)
10: return C
```

$$T(n) = 8T(n/2) + n^2$$

Matrix Multiplication with Divide & Conquer

$T(n) = 8T(n/2) + n^2$

Case 1: $d = 2 < \log_2 8 = 3$, so $T(n) = \Theta(n^3)$

$$T(n) = aT(n/b) + \Theta(n^d)$$

#subproblems

time per subproblem

time for divide and combine

Master Theorem (v2)

1. If $d < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $d = \log_b a$, then $T(n) = \Theta\left(n^d \lg n\right)$.
3. If $d > \log_b a$, then $T(n) = \Theta\left(n^d\right)$.

Strassen's Method

The key to a faster method is to make the recursion tree less bushy.*

In 1969, Volker Strassen published a remarkable algorithm for matrix multiplication with sub-cubic running time.

Four (magic) steps:

1. Partition each A, B and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices

2. Create ten matrices $S_1, S_2, \ldots, S_{10}$, each is of size $\frac{n}{2} \times \frac{n}{2}$ and is the sum or difference of two matrices created in step 1.

3. Using the matrices of step 1 and 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$ each of size $\frac{n}{2} \times \frac{n}{2}$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices.

*[Cormen p. 79]

Strassen's Method

Step 1. Partition each A, B and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices

Step 2.

$$S_1 = B_{12} - B_{22},$$
$$S_2 = A_{11} + A_{12},$$
$$S_3 = A_{21} + A_{22},$$
$$S_4 = B_{21} - B_{11},$$
$$S_5 = A_{11} + A_{22},$$
$$S_6 = B_{11} + B_{22},$$
$$S_7 = A_{12} - A_{22},$$
$$S_8 = B_{21} + B_{22},$$
$$S_9 = A_{11} - A_{21},$$
$$S_{10} = B_{11} + B_{12}.$$

Step 3.

$$P_1 = A_{11} \times S_1,$$
$$P_2 = S_2 \times B_{22},$$
$$P_3 = S_3 \times B_{11},$$
$$P_4 = A_{22} \times S_4,$$
$$P_5 = S_5 \times S_6,$$
$$P_6 = S_7 \times S_8,$$
$$P_7 = S_9 \times S_{10}.$$

Step 4.

$$C_{11} = P_5 + P_4 - P_2 + P_6,$$
$$C_{12} = P_1 + P_2,$$
$$C_{21} = P_3 + P_4,$$
$$C_{22} = P_5 + P_1 - P_3 - P_7,$$

$$T(n) = 7T(n/2) + n^2 \quad O(n^{2.81})$$

Some Notes on Strassen's Method

Strassen's algorithms has caused much excitement when published, as most people believed that $O(n^3)$ is the limit.

It has led to more research, and to date the most efficient algorithm asymptotically is based on Coppersmith and Winograd's method with a running time of $O(n^{2.376})$.

In practice, Strassen's method has problems with numerical stability. It also suffers from large constant factors hidden in the asymptotic analysis, so standard multiplication is faster on small matrices.

It's still used in hybrid algorithms, where standard multiplication is only used for small enough subproblems in the recursion tree.

Longest Palindrome Subsequence

A palindrome is a nonempty string over some alphabet that reads the same forward and backward.

Examples: `racecar`, `aibophobia`, `step on no pets`

Task: Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string.

Example: For input string `character` the algorithm should return `carac`.

Solution: Revert input string and use the LCS algorithm.

Bitonic Euclidean TSP

We are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points.

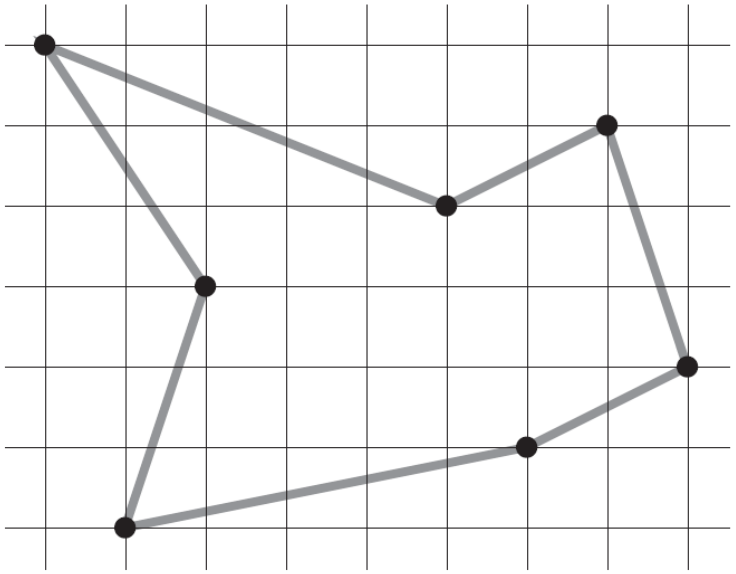
The general problem in intractable, see TSP.

Let's simplify the problem by restricting our attention to **bitonic tours**.

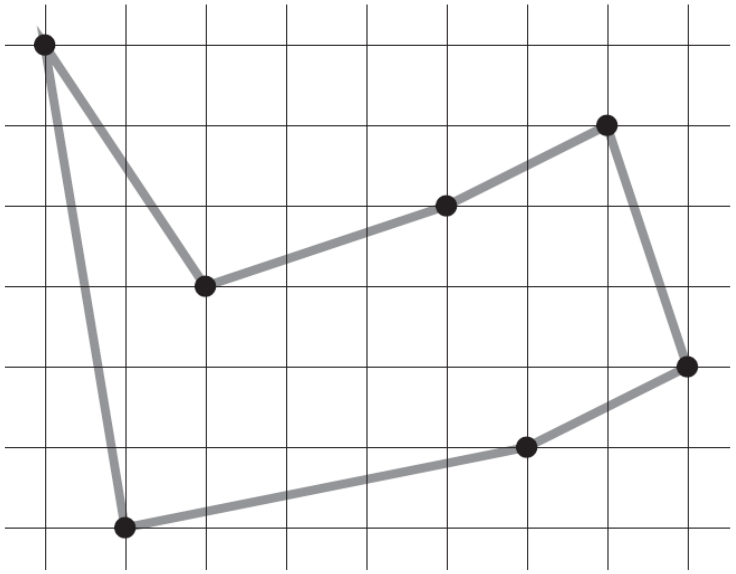
Bitonic Tours

A **bitonic tour** starts at the leftmost point, goes strictly rightward to the rightmost point, and then goes strictly leftward back to the starting point.

Bitonic tours can be computed in polynomial time.



shortest tour, not bitonic



shortest bitonic tour

Bitonic Tours

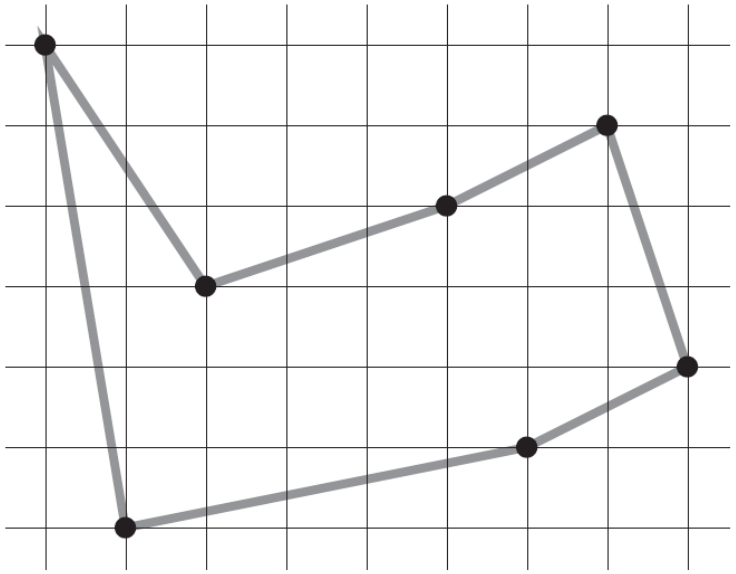
A **bitonic tour** starts at the leftmost point, goes strictly rightward to the rightmost point, and then goes strictly leftward back to the starting point.

Bitonic tours can be computed in polynomial time.

Task: Devise an algorithm for computing an optimal bitonic tour.

You may assume that no points have the same x-coordinate.

Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.

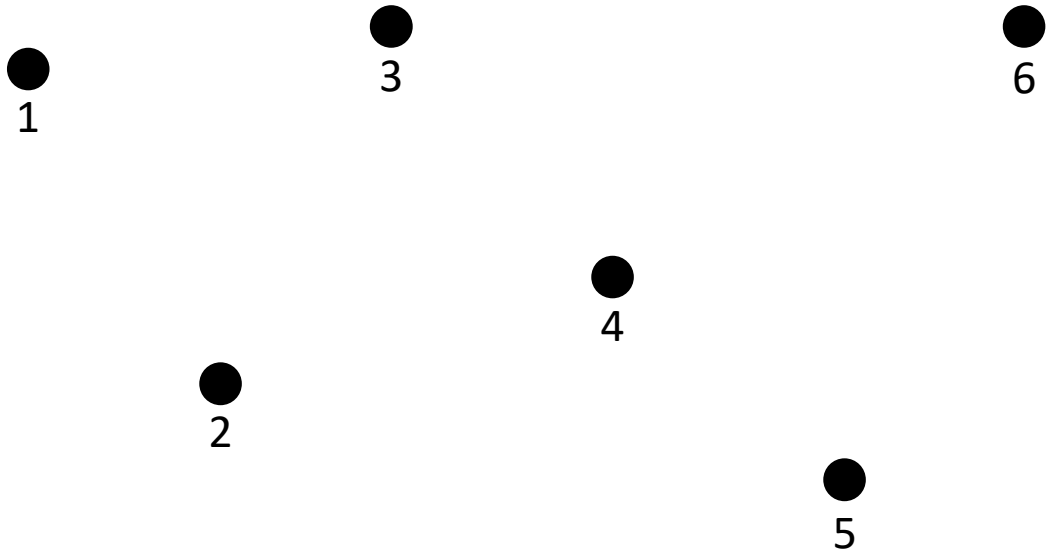


shortest bitonic tour

Bitonic Tours

Sort the points by x-coordinate. So we get a list of points $\langle p_1, p_2, \ldots, p_n \rangle$ with p_1 being the leftmost, p_n the rightmost point.

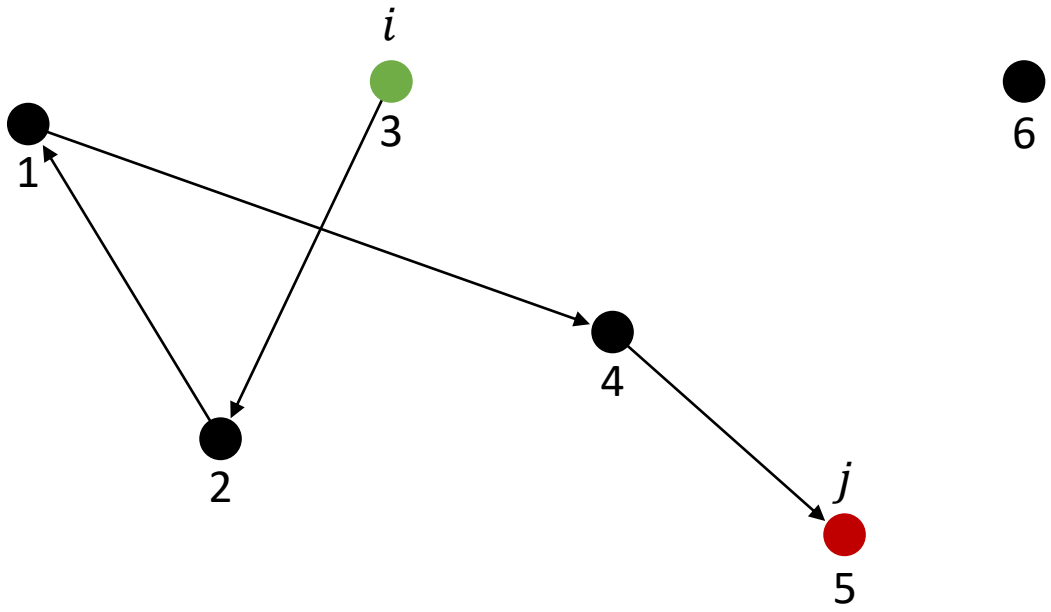
We define subproblems of the following form: a bitonic path P_{ij} where $i \leq j$ includes all points $p_1, p_2, \ldots, p_j$; it starts at p_i, goes strictly left to point p_1 and then strictly right to p_j.



Bitonic Tours

Sort the points by x-coordinate. So we get a list of points $\langle p_1, p_2, \ldots, p_n \rangle$ with p_1 being the leftmost, p_n the rightmost point.

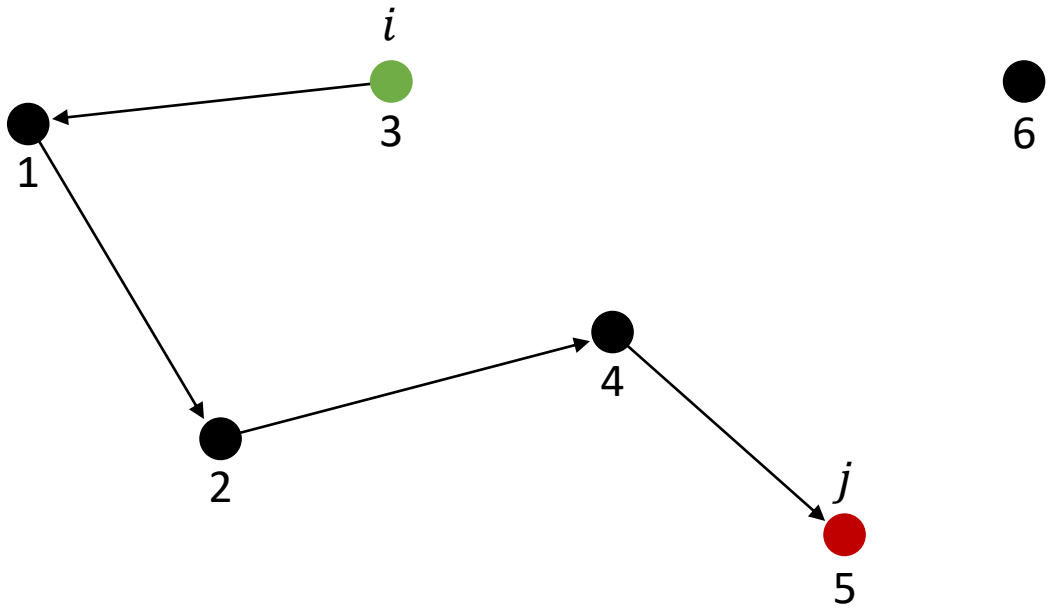
We define subproblems of the following form: a bitonic path P_{ij} where $i \leq j$ includes all points $p_1, p_2, \ldots, p_j$; it starts at p_i, goes strictly left to point p_1 and then strictly right to p_j.



Bitonic Tours

Sort the points by x-coordinate. So we get a list of points $\langle p_1, p_2, \ldots, p_n \rangle$ with p_1 being the leftmost, p_n the rightmost point.

We define subproblems of the following form: a bitonic path P_{ij} where $i \leq j$ includes all points $p_1, p_2, \ldots, p_j$; it starts at p_i, goes strictly left to point p_1 and then strictly right to p_j.



Bitonic Tours

Sort the points by x-coordinate. So we get a list of points $\langle p_1, p_2, \ldots, p_n \rangle$ with p_1 being the leftmost, p_n the rightmost point.

We define subproblems of the following form: a bitonic path P_{ij} where $i \leq j$ includes all points $p_1, p_2, \ldots, p_j$; it starts at p_i, goes strictly left to point p_1 and then strictly right to p_j.

Let $b[i,j]$ be the length of the shortest bitonic path P_{ij}.

We have the following formulation of $b[i,j]$ for $1 \leq i \leq j \leq n$.

$b[1,2] = d(p_1, p_2)$, Euclidean distance

$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j)$ for $i < j-1$

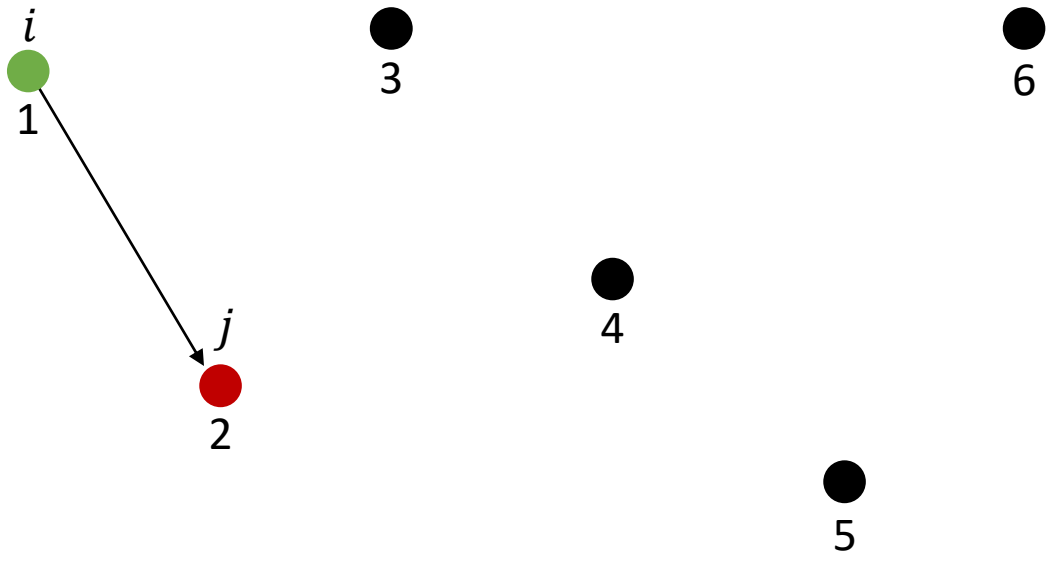
$b[j-1,j] = \min_{1 \leq k < j-1} \{b[k,j-1] + d(p_k, p_j)\}$.

Bitonic Tours

$b[1,2] = d(p_1, p_2)$,

$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j)$ for $i < j-1$

$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}$.

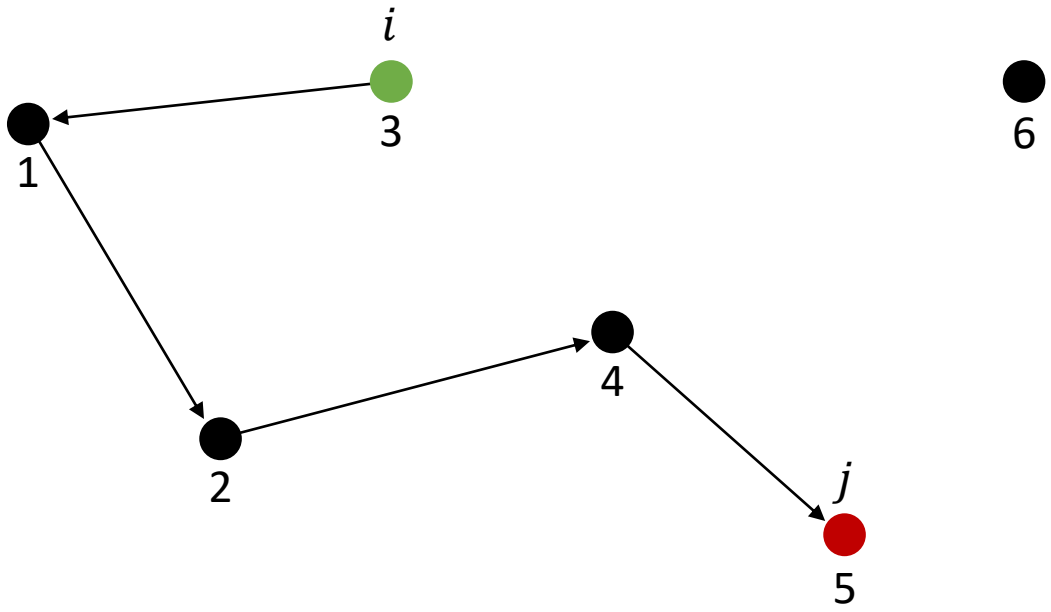


Bitonic Tours

$$b[1,2] = d(p_1, p_2),$$

$$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j) \text{ for } i < j-1$$

$$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}.$$

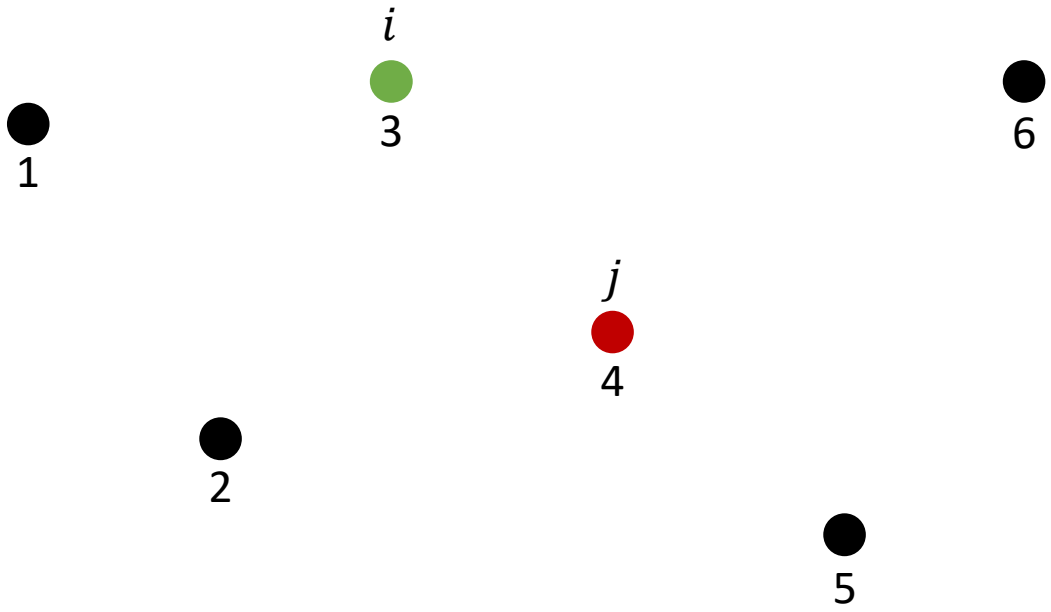


Bitonic Tours

$$b[1,2] = d(p_1, p_2),$$

$$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j) \text{ for } i < j-1$$

$$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}.$$

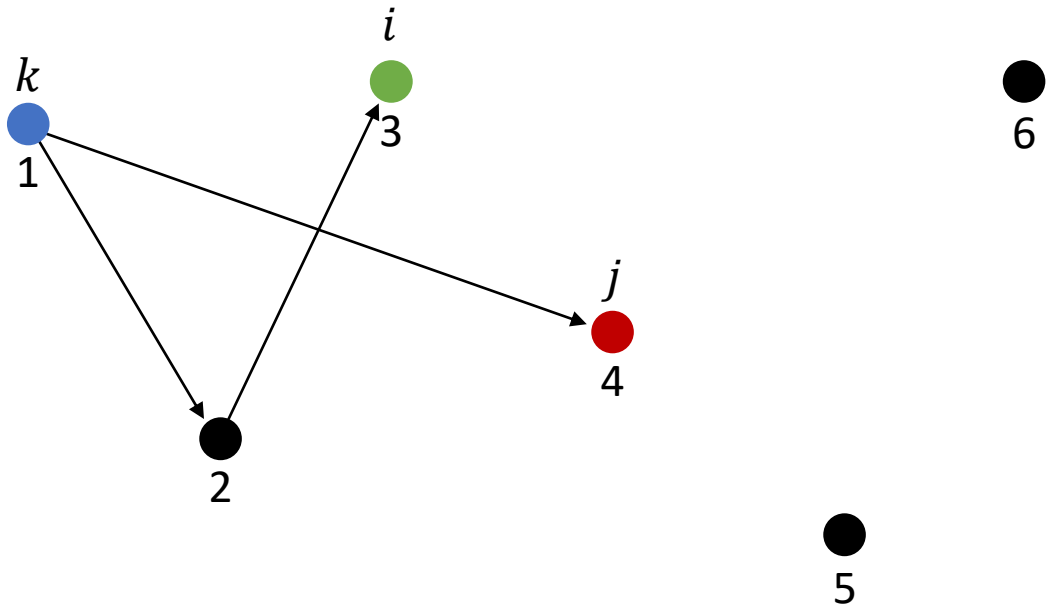


Bitonic Tours

$$b[1,2] = d(p_1, p_2),$$

$$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j) \text{ for } i < j-1$$

$$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}.$$

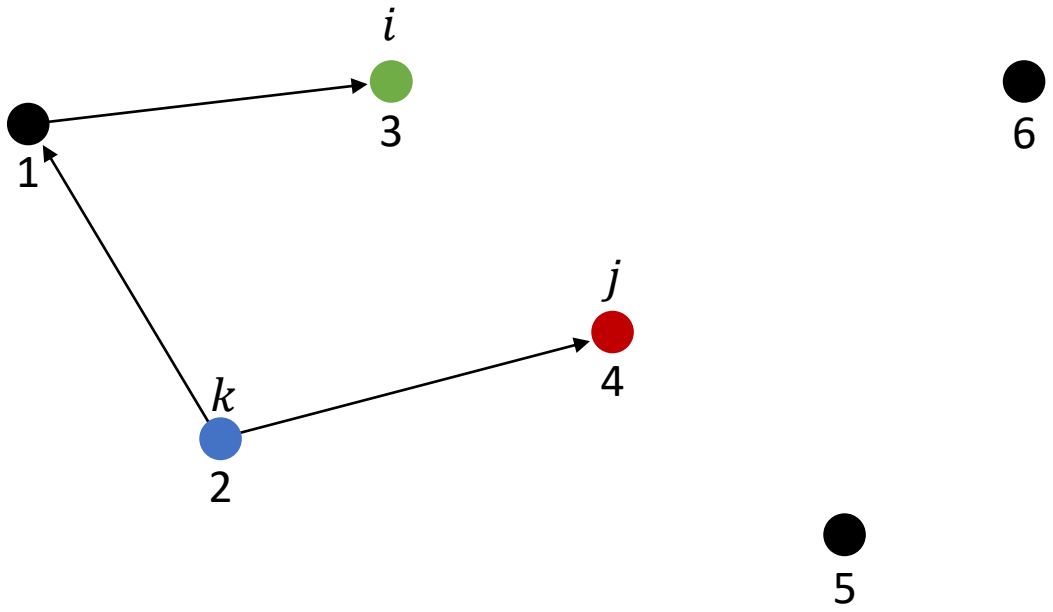


Bitonic Tours

$$b[1,2] = d(p_1, p_2),$$

$$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j) \text{ for } i < j-1$$

$$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}.$$



Bitonic Tours

$$b[1,2] = d(p_1, p_2),$$

$$b[i,j] = b[i,j-1] + d(p_{j-1}, p_j) \text{ for } i < j-1$$

$$b[j-1,j] = \min_{1 \le k < j-1} \{b[k,j-1] + d(p_k, p_j)\}.$$

$$b[n,n] = b[n-1,n] + d(p_{n-1}, p_n).$$

Shortest Bitonic Tour

```
BITONIC-TSP(p)
 1: sort the points in order of increasing x-coordinate
 2: b[1,2] = d(p[1],p[2])
 3: for j = 3 to n
 4:     for i = 1 to j-2
 5:         b[i,j] = b[i,j-1] + d(p[j-1],p[j])
 6:         r[i,j] = j-1
 7:     b[j-1,j] = inf
 8:     for k = 1 to j-2
 9:         q = b[k,j-1] + d(p[k],p[j])
10:         if q < b[j-1,j]
11:             b[j-1,j] = q
12:             r[j-1,j] = k
13: b[n,n] = b[n-1,n] + d(p[n-1],p[n])
14: return b and r
```