Design and Analysis of Algorithms

2. Basics of algorithm design & analysis

Mingyu XIAO

School of Computer Science and Engineering
University of Electronic Science and Technology of China

2.1 Analysis

Consider the following problems:

1. What is the difference between polynomial time and exponential time?
2. How to evaluate the running time and space?
3. How to denote the running time or space?

How To Measure the Running Time

Use a clock to calculate the running time?
--May not be good.

We Hope: there is a uniform standard for all instances.
For most algorithms, the running time and space are related to the size of the input.

Different types of running times:

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

How To Measure the Running Time

There are also some other kinds of running times, such as smooth analysis and so on.

We only consider worst case running time in this course.

Questions:

Why do we consider worst case running time for most problems?

If the worst case running time is W, then there is at least one instance that the running time of it is W?

How To Measure the Running Time

Consider this problem: Max independent set problem
To find a vertex set of max cardinality in a given graph such that no pair of vertices in the set are adjacent.

A brute force method for this problem: For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

What is the worst case running time of this algorithm?

It takes 2^N time for inputs of size N (N vertices).
(any problem for this algorithm?)

A Famous Story

在席·盖莫夫著的《从一到无穷大》中记载着一个关于古印度舍罕王（Shirham）的故事：

舍罕王打算重赏“国际象棋”（真正的国际象棋是后来进化演变来的）的发明人和进贡者，宰相西萨·班·达伊尔（Sissa Ben Dahir）。这位聪明的大臣的胃口看来并不大，他跪在国王面前说：“陛下，请您在这张棋盘的第一个小格内，赏给我一粒麦子；在第二个小格内两粒，第三个给四粒，照这样下去，每个小格内都比以前一个小格加一倍。陛下啊，把这样摆满棋盘上所有64格的麦粒，都赏给您的仆人罢！”

舍罕王听后很生气，说：”你也太小看我们印度国了，我拿一袋子麦子全给你好了。“

西萨·班·达伊尔坚持说还是按要求放满好。最后大家拿了许多袋麦子过来都只完成了一小部分的填充工作，大家这才感觉到问题的严重。

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

What It Matters

Shirham's story.

The amount of wheat

0	1	2	3	……	23	……	63
2^0	2^1	2^2	2^3	……	2^{23}	……	2^{63}
1	2	4	8	……	8388608	……	9223372036854775808
Sum	3	7	15	……	16777215	……	18446744073709551615

It takes more than 2,000,000 years for a modern computer to count the wheat!

Brute force solution to the max impendent set problem.

If the graph has 100 vertices,

Number of solution: 2^{100}=

1267650600228229401496703205376

Polynomial and exponential (多项式和指数)

上表中我们可以观察到 n^3 和 3^n 有巨大差别。
这个差别就是**指数与多项式的差别**。

在n^3 和 3^n 中，如果n变成2n的话（也就是输入增大一倍），两个数分别是以前的多少倍?
对于 n^3 来说，是8倍，一个常数倍;
对于 3^n 来说，是3^n倍，这个倍数与n有关。

指数增长是恐怖的，在算法运行时间中应当尽量避免。很多问题很容易就可以给出一个指数运行时间的穷举搜索算法，但是否都存在多项式算法？ (**计算机科学的核心问题**)

是否每个多项式时间的算法都有效?

数量级的差别

对于一个输入为n的问题，现给出两个算法A和B.
算法A运行100n步，算法B运行nlogn步。
哪个算法好?

若算法A运行n^2+100n步，算法B运行$2n^2$步。
哪个算法好?

很多时候我们认为n和n^2之间的差异是较大的，而n和5n之间的差异是微小的甚至可以忽悠不计的，怎样表达这种差异?

用渐进表达式可以解决这个问题。

就算都是多项式，也需表现差异

渐进表达式

常数间的差异较小

很多情况下，没有意义去说一个运行$2n^2$ 步的算法比一个运行n^2 步的算法要好.

在渐进表式方式中我们忽略常数，这样我们将下面式子看成相等的:

$100n^2$，$1.5n^2$，n^2+4，$10n^2-3n+6$.

用如下符合表式 $\Theta(n^2)$.

渐进表式的核心内容就是忽略常数！

渐进表式是一个被大家认同的计算机里表式运行时间和空间的方法。

渐进表式系统产生另一个数学系统，与精确数学、模糊数学、近似数学都不相同。

2.2 Asymptotic Order of Growth (渐进分析)

渐近分析的符号

在下面的讨论中，对所有n，$f(n) \geq 0$，$g(n) \geq 0$。

（1）渐近上界记号O

$O(g(n)) = \{ f(n) \mid$ 存在正常数c和n_0使得对所有$n \geq n_0$有：$0 \leq f(n) \leq cg(n) \}$

（2）渐近下界记号Ω

$\Omega(g(n)) = \{ f(n) \mid$ 存在正常数c和n_0使得对所有$n \geq n_0$有：$0 \leq cg(n) \leq f(n) \}$

（3）紧渐近界记号Θ

$\Theta(g(n)) = \{ f(n) \mid$ 存在正常数c_1, c_2和n_0使得对所有$n \geq n_0$有：$c_1 g(n) \leq f(n) \leq c_2 g(n) \}$

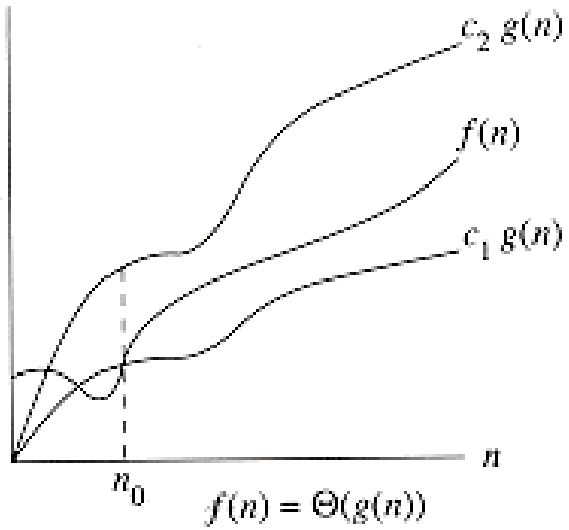
如果 f(n) 集合 O(g(n))中的一个成员，我们说f(n) 属于 O(g(n))。

例如： $f(n) = 32n^2 + 17n + 32$.

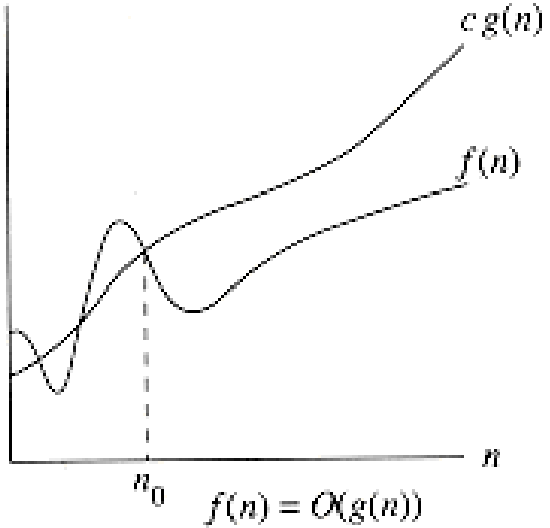
f(n) 属于 $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.

f(n) 不属于 $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

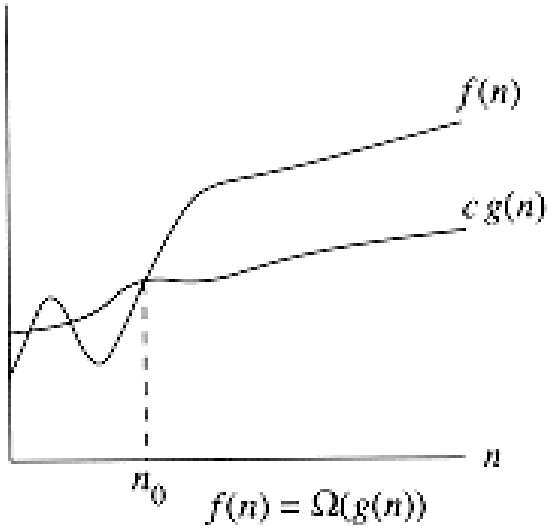
渐近分析的符号



(a)



(b)



(c)

$f(n)=\Theta(g(n))$

$\Updownarrow \cong$

$f(n)=g(n)$

$f(n)=O(g(n))$

$\Updownarrow \cong$

$f(n) \leq g(n)$

$f(n)= \Omega(g(n))$

$\Updownarrow \cong$

$f(n) \geq g(n)$

更多渐近分析的符号

在下面的讨论中，对所有n，$f(n) \geq 0$，$g(n) \geq 0$。

（4）非紧上界记号o

$o(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0 >0$使得对所有$n \geq n_0$有：

$0 \leq f(n) < cg(n) \}$

等价于 $f(n) / g(n) \to 0$ ， as $n \to \infty$。

（5）非紧下界记号ω

$\omega(g(n)) = \{ f(n) \mid$ 对于任何正常数$c>0$，存在正数和$n_0 >0$使得对所有$n \geq n_0$有：

$0 \leq cg(n) < f(n) \}$

等价于 $f(n) / g(n) \to \infty$ ， as $n \to \infty$。

$f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$

在渐进分析中等于符号的滥用

我们用f(n)= O(g(n))来表示 f(n) 是 O(g(n))的一个成员函数而不用传统的T(n)∈O(f(n))来表示。原因？没有原因，习惯问题而已。

例子

- $f(n) = 5n^3$; $g(n) = 3n^2$
- $f(n) = O(n^3) = g(n)$
- but $f(n) \neq g(n)$.

更好的表式方式: $T(n) \in O(f(n))$.

- $5n^3 \in O(n^3)$
- $3n^2 \in O(n^2) \subset O(n^3)$

在渐进分析中等于符号的滥用

因此$f(n)= \Theta(g(n))$的确切意义是：$f(n) \in \Theta(g(n))$。

一般情况下，等式和不等式中的渐近记号$\Theta(g(n))$表示$\Theta(g(n))$中的某个函数。

例如：$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ 表示

$2n^2 +3n +1=2n^2 + f(n)$，其中$f(n)$ 是$\Theta(n)$中某个函数。

等式和不等式中渐近记号O,o, Ω和ω的意义是类似的。

渐近分析中函数比较

$f(n) = O(g(n)) \approx a \leq b$;

$f(n) = \Omega(g(n)) \approx a \geq b$;

$f(n) = \Theta(g(n)) \approx a = b$;

$f(n) = o(g(n)) \approx a < b$;

$f(n) = \omega(g(n)) \approx a > b$.

渐近分析记号的若干性质

（1）传递性：

$f(n)=\Theta(g(n)),\quad g(n)=\Theta(h(n)) \Rightarrow f(n)=\Theta(h(n))$;

$f(n)=O(g(n)),\quad g(n)=O(h(n)) \Rightarrow f(n)=O(h(n))$;

$f(n)=\Omega(g(n)),\quad g(n)=\Omega(h(n)) \Rightarrow f(n)=\Omega(h(n))$;

$f(n)=o(g(n)),\quad g(n)=o(h(n)) \Rightarrow f(n)=o(h(n))$;

$f(n)=\omega(g(n)),\quad g(n)=\omega(h(n)) \Rightarrow f(n)=\omega(h(n))$;

（2）反身性：

$f(n)= \Theta(f(n))$;

$f(n)= O(f(n))$;

$f(n)= \Omega(f(n))$.

（3）对称性：

$f(n)= \Theta(g(n)) \Leftrightarrow g(n)= \Theta(f(n))$.

（4）互对称性：

$f(n)= O(g(n)) \Leftrightarrow g(n)= \Omega(f(n))$;

$f(n)= o(g(n)) \Leftrightarrow g(n)= \omega(f(n))$;

（5）算术运算：

$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$ ；

$O(f(n))+O(g(n)) = O(f(n)+g(n))$ ；

$O(f(n))*O(g(n)) = O(f(n)*g(n))$ ；

$O(cf(n)) = O(f(n))$ ；

$g(n)= O(f(n)) \Rightarrow O(f(n))+O(g(n)) = O(f(n))$ 。

规则$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$ 的证明：

对于任意$f_1(n) \in O(f(n))$，存在正常数c_1和自然数n_1，使得对所有$n \geq n_1$，有$f_1(n) \leq c_1 f(n)$。

类似地，对于任意$g_1(n) \in O(g(n))$，存在正常数c_2和自然数n_2，使得对所有$n \geq n_2$，有$g_1(n) \leq c_2 g(n)$。

令$c_3=\max\{c_1, c_2\}$，$n_3 =\max\{n_1, n_2\}$，$h(n)= \max\{f(n),g(n)\}$。

则对所有的 $n \geq n_3$，有

$$f_1(n) + g_1(n) \leq c_1 f(n) + c_2 g(n)$$
$$\leq c_3 f(n) + c_3 g(n) = c_3(f(n) + g(n))$$
$$\leq c_3 2 \max\{f(n),g(n)\}$$
$$= 2c_3 h(n) = O(\max\{f(n),g(n)\}) .$$

作业1：

证明：$O(f(n)+g(n)) = O(\max\{f(n),g(n)\})$，其中f(n)和g(n)是关于n的两个函数。

最常用的关系式

多项式. $a_0 + a_1 n + \ldots + a_d n^d = \Theta(n^d)$ 其中 $a_d > 0$.

对数. $O(\log_a n) = O(\log_b n)$ 其中 $a, b > 0$为常数.

对数. 对任意 $x > 0$, $\log n = O(n^x)$.

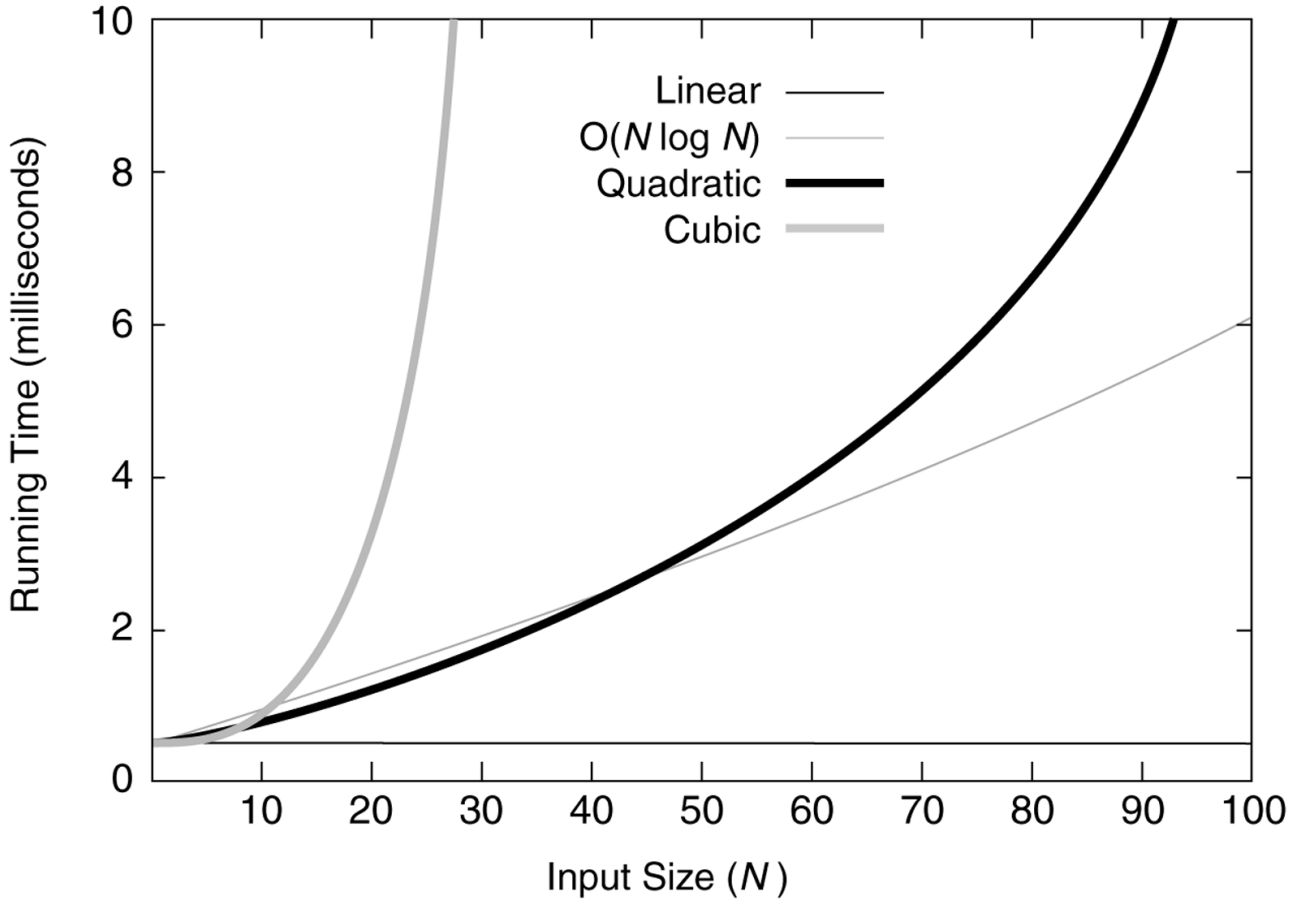
指数. 对任意 $r > 1$ 和 $d > 0$, $n^d = O(r^n)$.

重点记住内容!

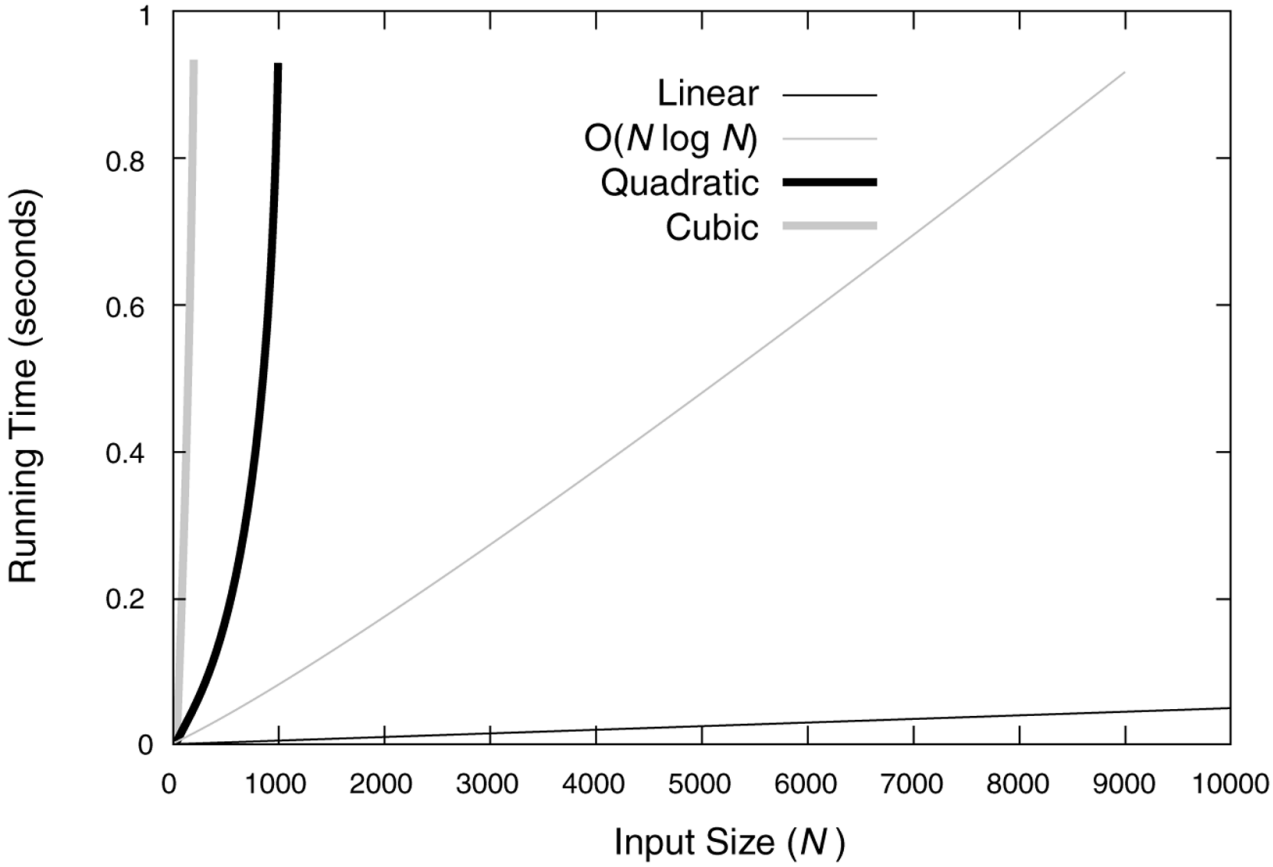
算法分析中常见的复杂性函数

FUNCTION	NAME
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

小规模数据


Linear
O(N log N)
Quadratic
Cubic
Running Time (milliseconds)
Input Size (N)
0
2
4
6
8
10
10
20
30
40
50
60
70
80
90
100

中等规模数据


Running Time (seconds)
1
0.8
0.6
0.4
0.2
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Input Size (N)
Linear
O(N log N)
Quadratic
Cubic

2.3 Three Techniques for Designing Algorithms

1. Greedy Algorithms
2. Divide and Conquer
3. Dynamical Programming

Greedy Algorithm

Basic Idea: In each iteration, we choose the "best" solution at that moment. This "best" solution may not yield the BEST final solution.

Local optimal v.s. Global optimal: Greedy algorithm is to make the locally optimal choice at each moment. In SOME problems, such strategy can lead to a global optimal solution.

Advantage: Simple, efficient

Disadvantage: May be incorrect / may not be optimal

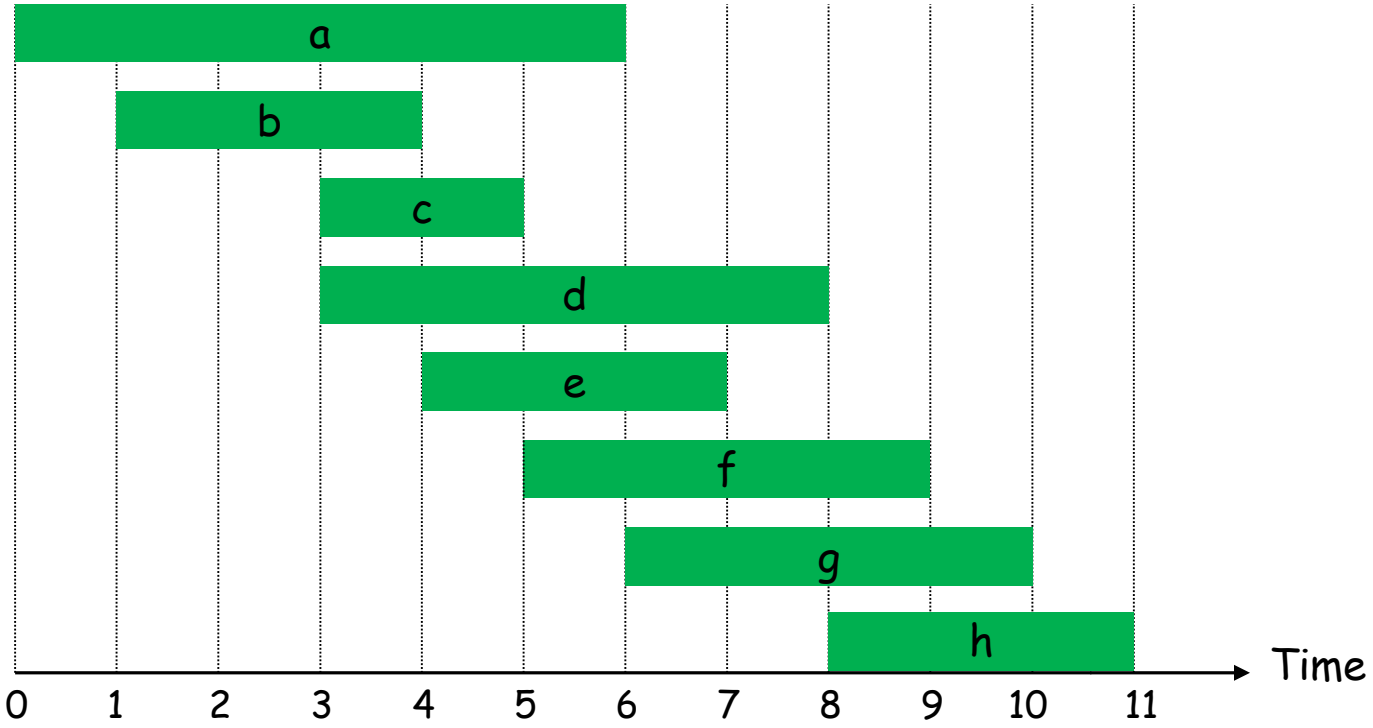
Interval Scheduling

Interval scheduling.

Job j starts at s_j and finishes at f_j.

Two jobs compatible if they don't overlap.

Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

[Earliest start time] Consider jobs in ascending order of start time s_j.

[Earliest finish time] Consider jobs in ascending order of finish time f_j.

[Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.

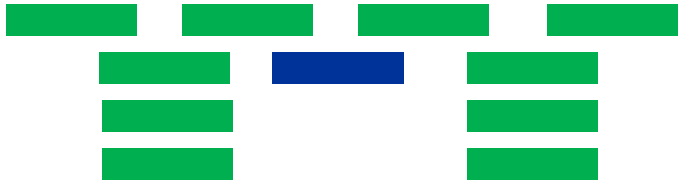
[Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval



breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

                jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n).

Remember job j* that was added last to A.

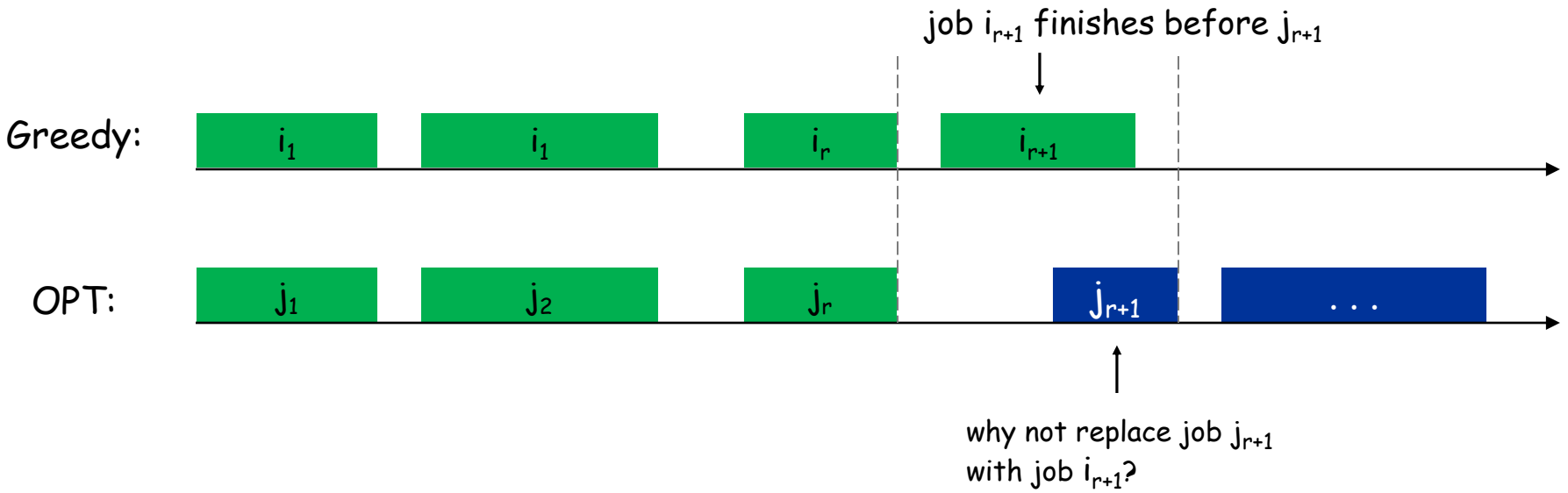
Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

Assume greedy is not optimal, and let's see what happens.
Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

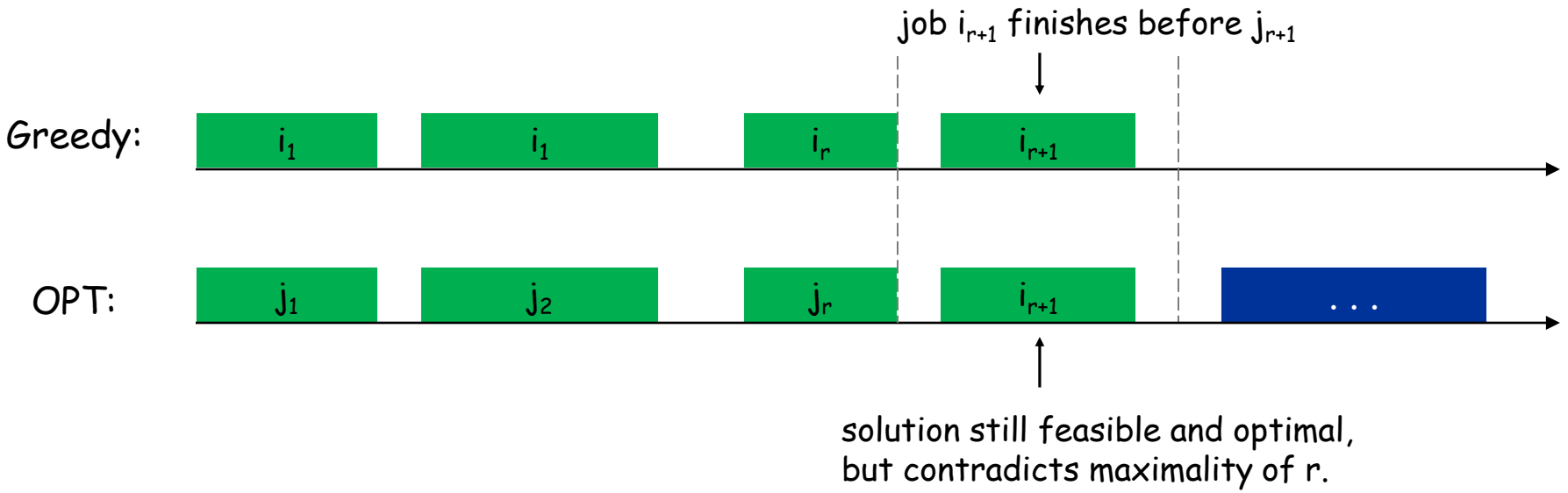


Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

Assume greedy is not optimal, and let's see what happens.
Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Other greedy algorithms. Kruskal, Prim, Dijkstra, Huffman, ...

Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

Ex: 34¢.

Dose this algorithm work? At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Coin-Changing

Observation. Greedy algorithm is not optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

Greedy: 100, 34, 1, 1, 1, 1, 1, 1.

Optimal: 70, 70.

Divide-and-Conquer

Divide-and-conquer.

Break up problem into several parts.
Solve each part recursively.
Combine solutions to sub-problems into overall solution.

Most common usage.

Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
Solve two parts recursively.
Combine two solutions into overall solution in linear time.

Integer Arithmetic

Add. Given two n-digit integers a and b, compute a + b.
O(n) bit operations.

Multiply. Given two n-digit integers a and b, compute a × b.
Brute force solution: $\Theta(n^2)$ bit operations.

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Add

Multiply

					1	1	0	1	0	1	0	1			
				*	0	1	1	1	1	1	0	1			
				1	1	0	1	0	1	0	1	0			
			0	0	0	0	0	0	0	0	0				
		1	1	0	1	0	1	0	1	0					
	1	1	0	1	0	1	0	1	0						
1	1	0	1	0	1	0	1	0							
1	1	0	1	0	1	0	1	0							
0	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	1	0	

Divide-and-Conquer Multiplication: Warmup

To multiply two n-digit integers:

Multiply four $\frac{1}{2}$n-digit integers.

Add two $\frac{1}{2}$n-digit integers, and shift to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0
\end{aligned}
$$

$$
T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)
$$

assumes n is a power of 2

Karatsuba Multiplication

To multiply two n-digit integers:

Add two $\frac{1}{2}n$ digit integers.

Multiply three $\frac{1}{2}n$-digit integers.

Add, subtract, and shift $\frac{1}{2}n$-digit integers to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \big((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0\big) + x_0 y_0
\end{aligned}
$$

A B A C C

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

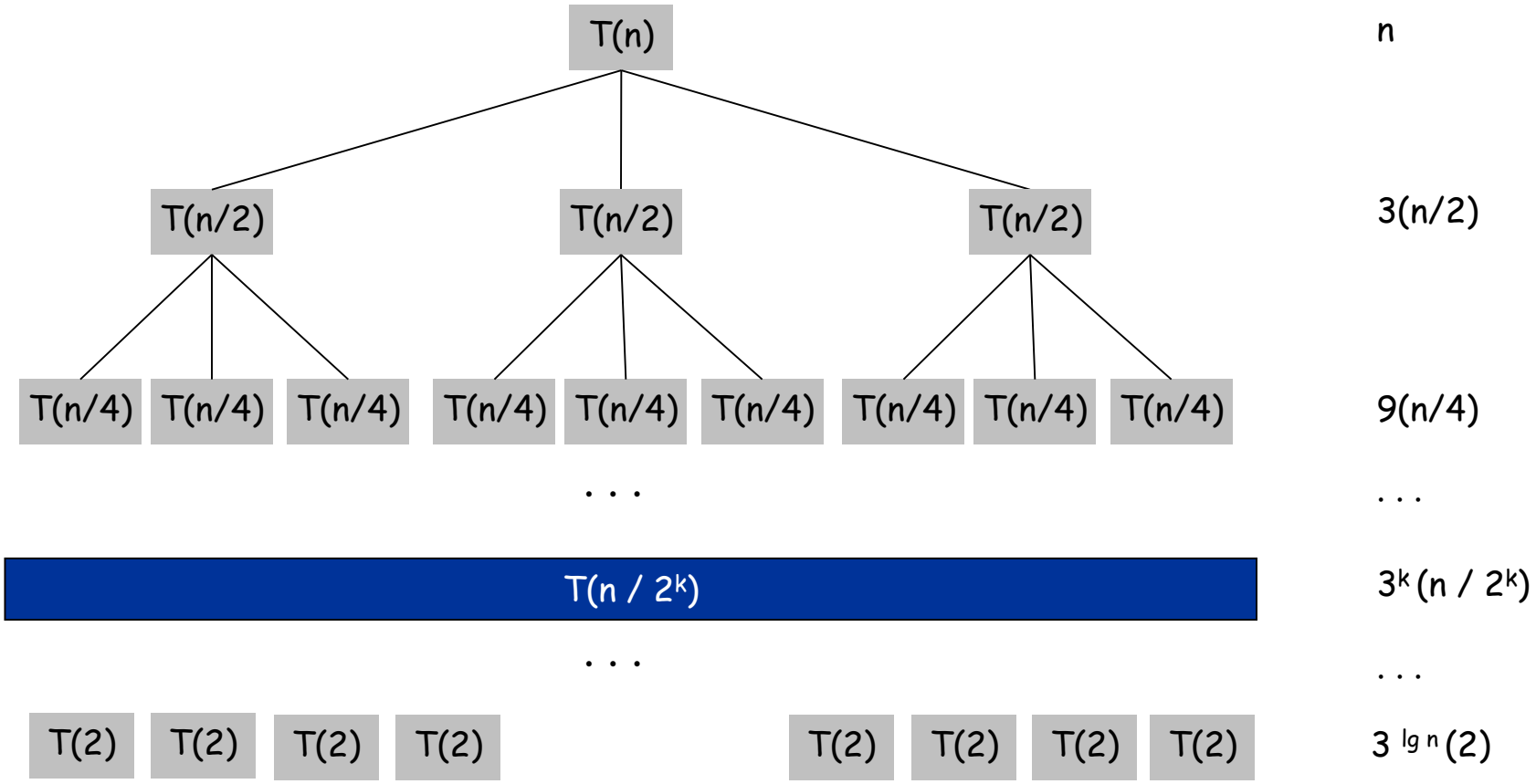
$$
T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}
$$

$$
\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})
$$

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n \left(\tfrac{3}{2}\right)^k = \frac{\left(\frac{3}{2}\right)^{1+\log_2 n} - 1}{\frac{3}{2} - 1} = 3n^{\log_2 3} - 2$$



Exercise: Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute C = AB.

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

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Warmup

Divide-and-conquer.

Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.

Conquer: multiply 8 $\frac{1}{2}n$-by-$\frac{1}{2}n$ recursively.

Combine: add appropriate products using 4 matrix additions.

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

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

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

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

7 multiplications.
18 = 10 + 8 additions (or subtractions).

Fast Matrix Multiplication

Fast matrix multiplication. (Strassen, 1969)

Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.

Compute: 14 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via 10 matrix additions.

Conquer: multiply 7 $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices recursively.

Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

Assume n is a power of 2.

T(n) = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Read by Yourself

How to solve recurrences?

Some methods:

- Substitution method
- Recursion-tree method
- Master method

Refer to 'Introduction to Algorithms' or other books of special topical on Recursion Theory.

Algorithmic Paradigms

Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into some sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Overlapping Subproblems

When a recursive algorithm re-visits the same problem over and over again, we say that the problem has overlapping subproblems.

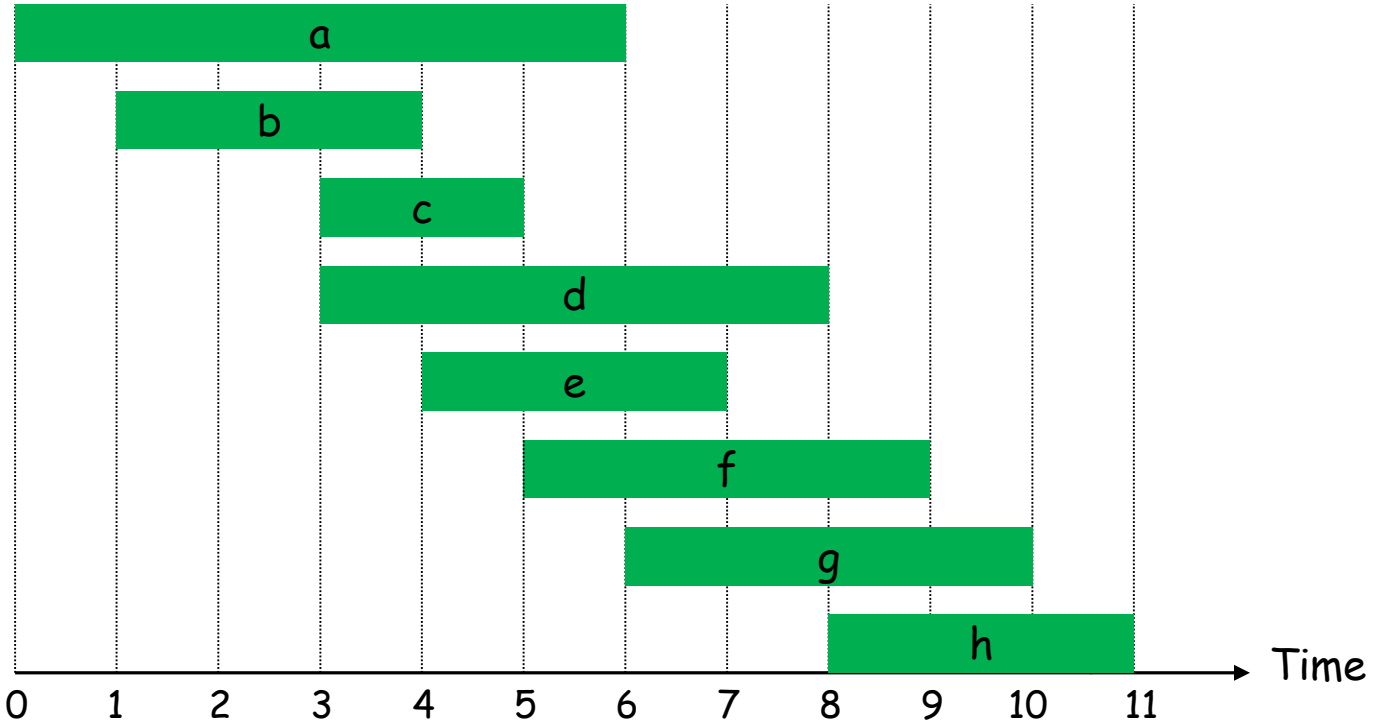
An idea to save the running time is to avoid computing the same subproblem twice.

This idea is the essential of dynamic programming.

Weighted Interval Scheduling

Weighted interval scheduling problem.

Job j starts at s_j, finishes at f_j, and has weight or value v_j .
Two jobs compatible if they don't overlap.
Goal: find maximum weight subset of mutually compatible jobs.



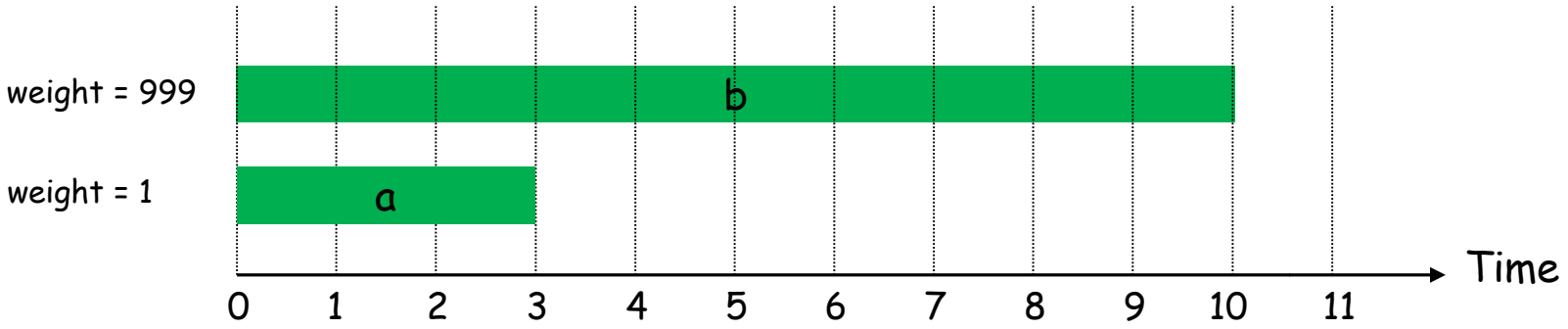
Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

Consider jobs in ascending order of finish time.

Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

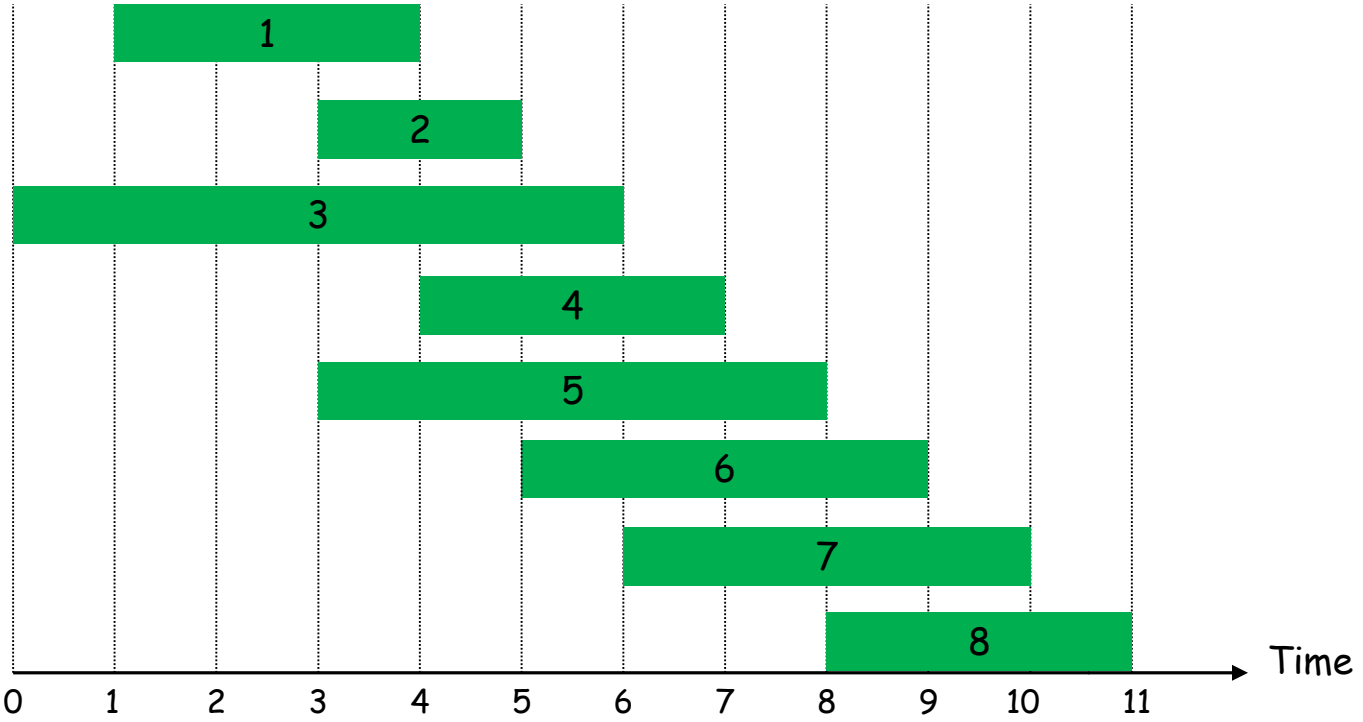


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.
- can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

Case 2: OPT does not select job j.
- must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

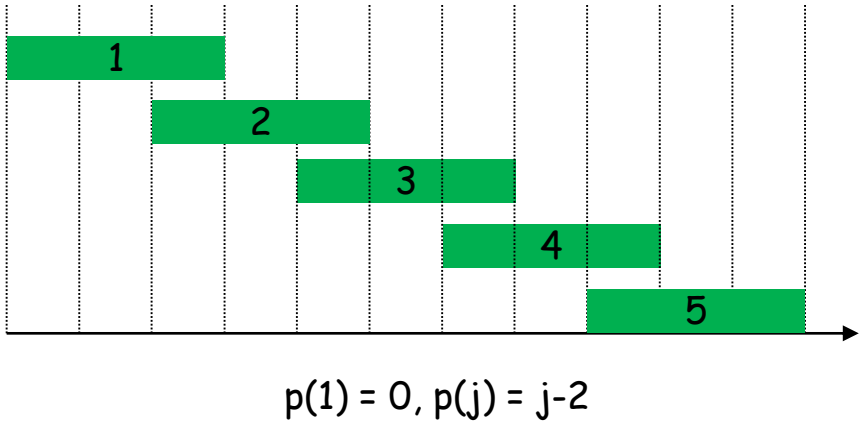
Brute force algorithm.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

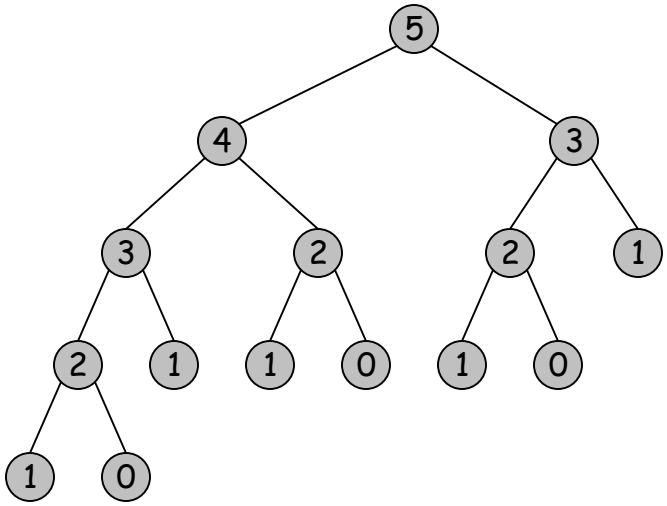
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty   ← global array
M[0] = 0

M-Compute-Opt(n) {
   if (M[n] is empty)
      M[n] = max(vn + M-Compute-Opt(p(n)), M-Compute-Opt(n-1))
   return M[n]
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

Sort by finish time: $O(n \log n)$.

Computing $p(\cdot)$: $O(n)$ after sorting by start time.

`M-Compute-Opt(j)`: each invocation takes $O(1)$ time and either

- (i) returns an existing value `M[j]`
- (ii) fills in one new entry `M[j]` and makes two recursive calls

Progress measure Φ = # nonempty entries of `M[]`.

- initially $\Phi = 0$, throughout $\Phi \leq n$.
- (ii) increases Φ by 1 $\Rightarrow$ at most n recursive calls.

Overall running time of `M-Compute-Opt(n)` is $O(n)$. ▪

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

Dynamic Programming

Recall that the main idea of dynamic programming is to save the running time by avoiding computing same subproblems.

What should be most important step of designing dynamic programming algorithms?

To find a good way to separate the problem into may overlapping subproblems.

Knapsack Problem

Knapsack problem.

Given n objects and a "knapsack."

Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.

Knapsack has capacity of W kilograms.

Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

Case 1: OPT does not select item i.
- OPT selects best of { 1, 2, …, i-1 }

Case 2: OPT selects item i.
- accepting item i does not immediately imply that we will have to reject other items
- without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Case 1: OPT does not select item i.
- OPT selects best of { 1, 2, …, i-1 } using weight limit w

Case 2: OPT selects item i.
- new weight limit = $w - w_i$
- OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

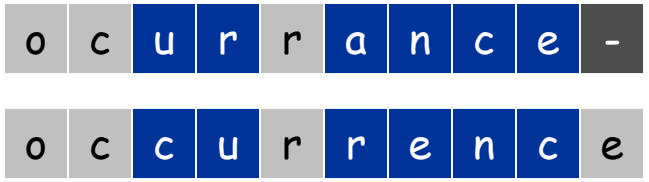
Sequence Alignment

String Similarity

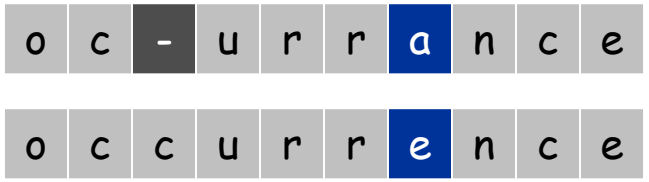
How similar are two strings?

ocurrance

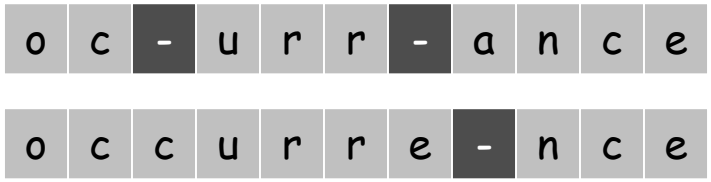
occurrence



5 mismatches, 1 gap



1 mismatch, 1 gap



0 mismatches, 3 gaps

Edit Distance

Applications.

Basis for Unix diff.

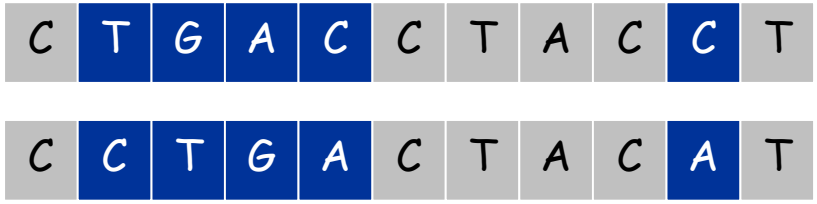
Speech recognition.

Computational biology.

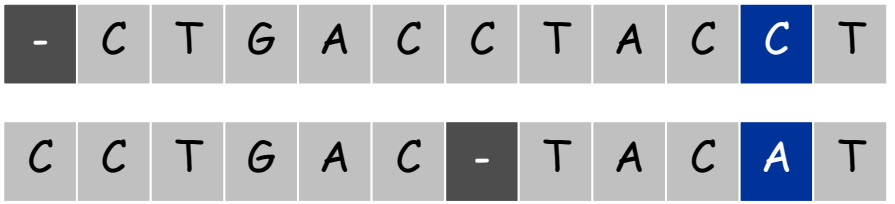
Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

Gap penalty δ; mismatch penalty α_{pq}.

Cost = sum of gap and mismatch penalties.



$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$



$2\delta + \alpha_{CA}$

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

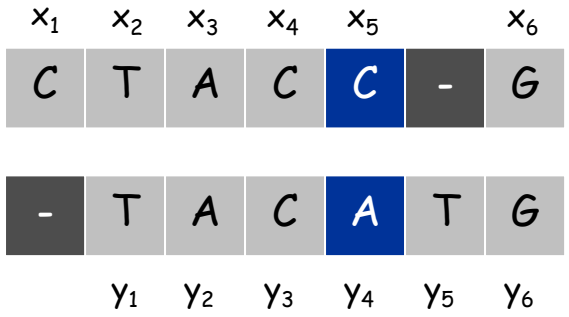
Def. An alignment M is a set of ordered pairs x_i-y_j such that each item occurs in at most one pair and no crossings.

Def. The pair x_i-y_j and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG vs. TACATG.

Sol: $M = x_2$-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.



Sequence Alignment: Problem Structure

Def. OPT(i, j) = min cost of aligning strings $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_j$.

Case 1: OPT matches x_i-y_j.

- pay mismatch for x_i-y_j + min cost of aligning two strings $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_{j-1}$

Case 2a: OPT leaves x_i unmatched.

- pay gap for x_i and min cost of aligning $x_1\ x_2\ \ldots\ x_{i-1}$ and $y_1\ y_2\ \ldots\ y_j$

Case 2b: OPT leaves y_j unmatched.

- pay gap for y_j and min cost of aligning $x_1\ x_2\ \ldots\ x_i$ and $y_1\ y_2\ \ldots\ y_{j-1}$

$$OPT(i,\ j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1,\ j-1) \\ \delta + OPT(i-1,\ j) \\ \delta + OPT(i,\ j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100{,}000$. 10 billions ops OK, but 10GB array?