

Algorithm Design Techniques

Chapter 10 of textbook 1

Algorithm Design

To now, we have examined a number of data structures and algorithms to manipulate them

We have seen examples of efficient strategies

- Divide and conquer
 - Merge sort
 - Quicksort
 - 3^N Calculation
- Greedy algorithms
 - Prim's algorithm
 - Kruskal's algorithm
 - Dijkstra's algorithm

Algorithm Design

We will now examine a number of strategies which may be used in the design of algorithms, including:

- Greedy algorithms
- Divide-and-conquer algorithms
- Dynamic programming

Algorithm Design

When searching for a solution, we may be interested in two types:

- Either we are looking for the optimal solution, or,
- We are interested in a solution which is **good enough**, where good enough is defined by a set of parameters

Algorithm Design

For many of the strategies we will examine, there will be certain circumstances where the strategy can be shown to result in an optimal solution

In other cases, the strategy may not be guaranteed to do so well

Algorithm Design

Any problem may usually be solved in multiple ways

The simplest to implement is ***brute force***

- We consider all possible solutions, and find that solution which is optimal

Algorithm Design

Brute force techniques often take too much time to run

We may use brute-force techniques to show that solutions found through other algorithms are either optimal or close-to-optimal

Algorithm Design

With brute force, we consider all possible solutions

Most other techniques build solutions, thus, we require the following definitions

Definition:

- A ***partial solution*** is a solution to a problem which could possibly be extended
- A ***feasible solution*** is a solution which satisfies any given requirements

Algorithm Design

Thus, we would say that a **brute-force search** tests all feasible solutions

Most techniques will build **feasible solutions from partial solutions** and thereby test only a subset of all possible feasible solutions

Algorithm Design

It may be possible in some cases to have partial solutions which are acceptable (that is, feasible) solutions to the problem

In other cases, partial solutions may be unacceptable, and therefore we must continue until we reach a feasible solution

Algorithm Design

We will look at two problems:

- the first requires an exact (optimal) solution
 - Examples: sudoku, finding password
- the second requires only an approximately optimal solution
 - Examples: Optimization

In the second case, it would be desirable, but not necessary, to find the optimal solution

Example

For example, consider the game of Sudoku

The rules are:

- each number must appear once in each row, column, and 3×3 outlined square

You are given some initial numbers, and if they are chosen appropriately, there is a unique solution

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

Example

Using brute force, we could try every possible solution, and discard those which do not satisfy the conditions

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

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

This technique would require us to check $9^{61} \approx 1.6 \times 10^{58}$ possible solutions

Greedy Algorithm

Definition

A greedy algorithm is an algorithm which has:

- A set of partial solutions from which a solution is built
- An *objective function* which assigns a value to any partial solution

Then given a partial solution, we

- Consider possible extensions of the partial solution
- Discard any extensions which are not feasible
- Choose that extension which minimizes the object function

This continues until some criteria has been reached

Optimal example

Prim's algorithm is a greedy algorithm:

- Any connected sub-graph of k vertices and $k - 1$ edges is a partial solution
- The value to any partial solution is the sum of the weights of the edges

Then given a partial solution, we

- Add that edge which does not create a cycle in the partial solution and which minimizes the increase in the total weight
- We continue building the partial solution until the partial solution has n vertices
- An optimal solution is found

Optimal example

Dijkstra's algorithm is a greedy algorithm:

- A subset of k vertices and known the minimum distance to all k vertices is a partial solution

Then given a partial solution, we

- Add that edge which is smallest which connects a vertex to which the minimum distance is known and a vertex to which the minimum distance is not known
- We define the distance to that new vertex to be the distance to the known vertex plus the weight of the connecting edge
- We continue building the partial solution until either:
 - The minimum distance to a specific vertex is known, or
 - The minimum distance to all vertices is known
- An optimal solution is found

No optimal example

In some cases, it may be possible that not even a optimal solution is found (big disadvantage)

- Consider the following greedy algorithm for solving Sudoku:
- For each empty square, starting at the top-left corner and going across:
 - Fill that square with the smallest number which does not violate any of our conditions
 - All feasible solutions have equal weight

Unfeasible example

Let's try this example the previously seen Sudoku square:

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

Unfeasible example

Neither 1 nor 2 fits into the first empty square, so we fill it with 3

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

Unfeasible example

The second empty square may be filled with 1

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

Unfeasible example

And the 3rd empty square may be filled with 4

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

Unfeasible example

At this point, we try to fill in the 4th empty square

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

Unfeasible example

Unfortunately, all nine numbers 1 – 9 already appear in such a way to block it from appearing in that square

- There is no known greedy algorithm which finds the one feasible solution

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

Near-optimal algorithms

We have seen:

- Prim's and Dijkstra's algorithms which are greedy and find the optimal solution
- A naïve greedy algorithm which attempts (and fails) to solve Sudoku

Next, we will see a greedy algorithm which finds a feasible, but not necessarily an optimal solution

Project management 0/1 knapsack problem

Situation:

- The next cycle for a given product is 26 weeks
- We have ten possible projects which could be completed in that time, each with an expected number of weeks to complete the project and an expected increase in revenue

This is also called the 0/1 knapsack problem

- You can place n items in a knapsack where each item has a value in rupees and a weight in kilograms
- The knapsack can hold a maximum of m kilograms

Project management 0/1 knapsack problem

Objective:

- As project manager, choose those projects which can be completed in the required amount of time which **maximizes revenue**

Project management 0/1 knapsack problem

The projects:

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)
A	15	210
B	12	220
C	10	180
D	9	120
E	8	160
F	7	170
G	5	90
H	4	40
J	3	60
K	1	10

Project management 0/1 knapsack problem

Let us first try to find an optimal schedule by trying to be as productive as possible during the 26 weeks:

- we will start with the projects in order from most time to least time, and at each step, select the longest-running project which does not put us over 26 weeks
- we will be able to fill in the gaps with the smaller projects

Project management 0/1 knapsack problem

Greedy-by-time (make use of all 26 wks):

- Project A: 15 wks
- Project C: 10 wks
- Project J: 1 wk

Total time: 26 wks

Expected revenue:
\$400 000

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)
A	15	210
B	12	220
C	10	180
D	9	120
E	8	160
F	7	170
G	5	90
H	4	40
I	3	60
J	1	10

Project management 0/1 knapsack problem

Next, let us attempt to find an optimal schedule by starting with the most :

- we will start with the projects in order from most time to least time, and at each step, select the longest-running project which does not put us over 26 weeks
- we will be able to fill in the gaps with the smaller projects

Project management 0/1 knapsack problem

Greedy-by-revenue (best-paying projects):

- Project B: \$220K
- Project C: \$180K
- Project H: \$ 60K
- Project K: \$ 10K

Total time: 26 wks

Expected revenue:
\$470 000

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)
B	12	220
A	15	210
C	10	180
F	7	170
E	8	160
D	9	120
G	5	90
J	3	60
H	4	40
K	1	10

Project management 0/1 knapsack problem

Unfortunately, either of these techniques focuses on projects which have high projected revenues or high run times

What we really want is to be able to complete those jobs which pay the most per unit of development time

Thus, rather than using development time or revenue, let us calculate the expected revenue per week of development time

Project management 0/1 knapsack problem

This is summarized here:

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)	Revenue Density (\$ / wk)
A	15	210	14 000
B	12	220	18 333
C	10	180	18 000
D	9	120	13 333
E	8	160	20 000
F	7	170	24 286
G	5	90	18 000
H	4	40	10 000
J	3	60	20 000
K	1	10	10 000

Project management 0/1 knapsack problem

Greedy-by-revenue-density:

- Project F: \$24 286/wk
- Project E: \$20 000/wk
- Project J: \$20 000/wk
- Project G: \$18 000/wk
- Project K: \$10 000/wk

Total time: 24 wks

Expected revenue:
\$490 000

Bonus: 2 weeks for bug fixing

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)	Revenue Density (\$/wk)
F	7	170	24 286
E	8	160	20 000
J	3	60	20 000
B	12	220	18 333
C	10	180	18 000
G	5	90	18 000
A	15	210	14 000
D	9	120	13 333
H	4	40	10 000
K	1	10	10 000

Project management 0/1 knapsack problem

Using brute force, we find that the optimal solution is:

- Project C: \$180 000
- Project E: \$170 000
- Project F: \$150 000
- Project K: \$ 10 000

Total time: 26 wks

Expected revenue:
\$520 000

Product ID	Completion Time (wks)	Expected Revenue (1000 \$)	Revenue Density (\$/wk)
A	15	210	14 000
B	12	220	18 333
C	10	180	18 000
D	9	120	13 333
E	8	160	20 000
F	7	170	24 286
G	5	90	18 000
H	4	40	10 000
J	3	60	20 000
K	1	10	10 000

Project management 0/1 knapsack problem

In this case, the greedy-by-revenue-density came closest to the optimal solution:

Algorithm	Expected Revenue
Greedy-by-time	\$400 000
Greedy-by-expected revenue	\$470 000
Greedy-by-revenue density	\$490 000
Brute force	\$520 000

- The run time is $\Theta(n \ln(n))$ — the time required to sort the list
- Later, we will see a dynamic program for finding an optimal solution with one additional constraint

Project management 0/1 knapsack problem

Of course, in reality, there are numerous other factors affecting projects, including:

- Flexible deadlines (if a delay by a week would result in a significant increase in expected revenue, this would be acceptable)
- Probability of success for particular projects
- The requirement for *banner* projects
 - Note that greedy-by-revenue-density had none of the larger projects

Quiz

Problem: You have to make a change of an amount using the smallest possible number of coins.

- Amount: \$18

Available coins are

- \$5 coin
- \$2 coin
- \$1 coin
- There is no limit to the number of each coin you can use.

5, 5, 5, 2, 1

Summary of greedy algorithms

We have seen the algorithm-design technique, namely greedy algorithms

- For some problems, appropriately-designed greedy algorithms may find either optimal or near-optimal solutions
- For other problems, greedy algorithms may a poor result or even no result at all

Their desirable characteristic is speed

Divide-and-Conquer Algorithm

Divide-and-conquer algorithms

We have seen four divide-and-conquer algorithms:

- Merge sort
- Quick sort
- 3^N Calculation

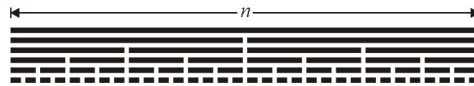
The steps are:

- A larger problem is broken up into smaller problems
- The smaller problems are recursively
- The results are combined together again into a solution

Divide-and-conquer algorithms

For example, merge sort:

- Divide a list of size n into $b = 2$ sub-lists of size $n/2$ entries
- Each sub-list is sorted recursively
- The two sorted lists are merged into a single sorted list



Divide-and-conquer algorithms

More formally, we will consider only those algorithms which:

- Divide a problem into b sub-problems, each approximately of size n/b
 - In merge sort, $b = 2$
 - In 3^N , $b = 2$
- Solve a ($1 \leq a \leq b$) of those sub-problems recursively
 - In merge sort, $a = 2$
 - In 3^N , $a = 1$
- Combine the solutions to the sub-problems to get a solution to the overall problem

Divide-and-conquer algorithms

With the three problems we have already looked at we have looked at two possible cases for $b = 2$:

Merge sort	$b = 2$	$a = 2$
3^N	$b = 2$	$a = 1$

Problem: the first two have different run times:

Merge sort	$\Theta(n \ln(n))$
3^N	$\Theta(\ln(n))$

Divide-and-conquer algorithms

Thus, just using a divide-and-conquer algorithm does not solely determine the run time

We must also consider

- The effort required to divide the problem into two sub-problems
- The effort required to combine the two solutions to the sub-problems

Divide-and-conquer algorithms

For merge sort:

- Division is quick (find the middle): $\Theta(1)$
- Merging the two sorted lists into a single list is a $\Theta(n)$ problem

For 3^N calculation:

- Division is also quick: $\Theta(1)$
- A return-from-function is preformed at the end which is $\Theta(1)$

Divide-and-conquer algorithms

Thus, we are able to write the expression as follows:

• 3^N :
 $\Theta(\ln(n))$

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + \Theta(1) & n > 1 \end{cases}$$

• Merge/quick sort:
 $\Theta(n \ln(n))$

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

In general, we will assume the work done combined work is of the form $O(n^k \log^p n)$

Divide-and-conquer algorithms

Thus, for a general divide-and-conquer algorithm which:

- Divides the problem into b sub-problems
- Recursively solves a of those sub-problems
- Requires $O(n^k \log^p n)$ work at each step requires

has a run time

$$T(n) = \begin{cases} 1 & n = 1 \\ a T\left(\frac{n}{b}\right) + O(n^k \log^p n) & n > 1 \end{cases}$$

Note: we assume a problem of size $n = 1$ is solved...

Master Theorem for Divide and Conquer

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

$a \geq 1$, $b > 1$, $k \geq 0$ and p is a real number

Case (1): $a > b^k$ then $T(n) = \theta(n^{\log_b a})$

Case(2): $a = b^k$:

if $p > -1$ then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

if $p = -1$ then $T(n) = \theta(n^{\log_b a} \log \log n)$

if $p < -1$ then $T(n) = \theta(n^{\log_b a})$

Case(3): $a < b^k$:

if $p \geq 0$ then $T(n) = \theta(n^k \log^p n)$

if $p = 0$ then $T(n) = O(n^k)$

Divide-and-conquer algorithms

Let us look at some more complex examples:

- Searching an ordered matrix
- Matrix multiplication

Searching an ordered matrix

Consider an $n \times n$ matrix where each row and column is linearly ordered; for example:

- How can we determine if 19 is in the matrix?

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Searching an ordered matrix

Consider the following search for 19:

- Search across until $a_{i,j+1} > 19$
- Alternate between
 - Searching down until $a_{i,j} > 19$
 - Searching back until $a_{i,j} < 19$

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

This requires us to check at most $3n$ entries: $O(n)$

Searching an ordered matrix

Can we do better than $O(n)$?

Logically, no: any number could appear in up to n positions, each of which must be checked

- Never-the-less: let's generalize checking the middle entry

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Searching an ordered matrix

$17 < 19$, and therefore, we can only exclude the top-left sub-matrix:

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Searching an ordered matrix

Thus, we must recursively search three of the four sub-matrices

- Each sub-matrix is approximately $n/2 \times n/2$

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Searching an ordered matrix

If the number we are searching for was less than the middle element, *e.g.*, 9, we would have to search three different squares

2	4	5	8	9	11	14
3	5	8	12	14	15	17
6	8	9	15	18	20	21
7	12	14	17	23	24	29
10	14	16	20	24	27	30
13	15	20	23	30	32	37
15	17	21	25	33	35	40

Searching an ordered matrix

Thus, the recurrence relation must be

$$T(n) = \begin{cases} 1 & n = 1 \\ 3T\left(\frac{n}{2}\right) + \Theta(1) & n > 1 \end{cases}$$

because

- $T(n)$ is the time to search a matrix of size $n \times n$
- The matrix is divided into 4 sub-matrices of size $n/2 \times n/2$
- Search 3 of those sub-matrices
- At each step, we only need compare the middle element: $\Theta(1)$

Searching an ordered matrix

Note that it is

$$T(n) = 3T(n/2) + \Theta(1) \Rightarrow T(n) = O(n^{\log_2 3})$$

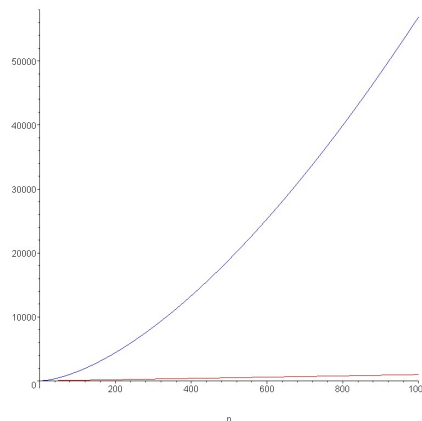
and **not**

$$T(n) = 3T(n/4) + \Theta(1)$$

We are breaking the $n \times n$ matrix into four $(n/2) \times (n/2)$ matrices

Searching an ordered matrix

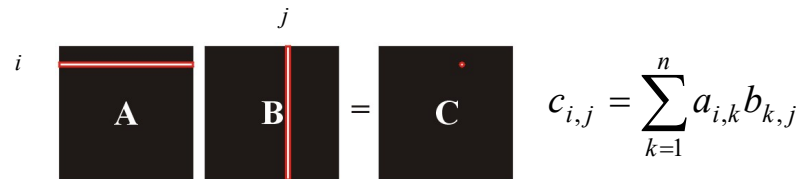
Therefore, this search is approximately $O(n^{1.585})$, which is significantly worse than a linear search:



Matrix multiplication

Consider multiplying two $n \times n$ matrices, $\mathbf{C} = \mathbf{AB}$

This requires the $\Theta(n)$ dot product of each of the n rows of \mathbf{A} with each of the n columns of \mathbf{B}



$$c_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}$$


The run time must be $\Theta(n^3)$

- Can we do better?

Matrix multiplication

In special cases, faster algorithms exist:

- If both matrices are diagonal or tri-diagonal $\Theta(n)$



- If one matrix is diagonal or tri-diagonal $\Theta(n^2)$



In general, however, this was not believed to be possible to do better

Matrix multiplication

Consider this product of two $n \times n$ matrices

- How can we break this down into smaller sub-problems?

$$\mathbf{A} \mathbf{B} = \mathbf{C}$$

Matrix multiplication

Break each matrix into four $(n/2) \times (n/2)$ sub-matrices

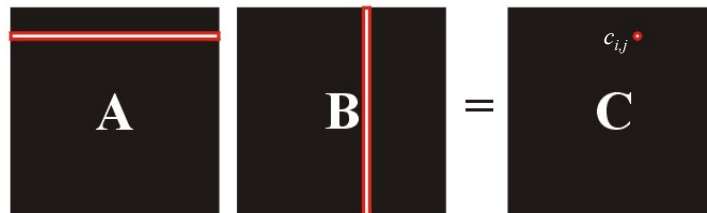
- Write each sub-matrix of \mathbf{C} as a sum-of-products

$$\begin{array}{cc|cc|cc} \mathbf{A} & & \mathbf{B} & & \mathbf{C} & \\ \hline \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \mathbf{A}_{0,0}\mathbf{B}_{0,0} + \mathbf{A}_{0,1}\mathbf{B}_{1,0} & \mathbf{A}_{0,0}\mathbf{B}_{0,1} + \mathbf{A}_{0,1}\mathbf{B}_{1,1} \\ \hline \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \mathbf{A}_{1,0}\mathbf{B}_{0,0} + \mathbf{A}_{1,1}\mathbf{B}_{1,0} & \mathbf{A}_{1,0}\mathbf{B}_{0,1} + \mathbf{A}_{1,1}\mathbf{B}_{1,1} \end{array} =$$

Matrix multiplication

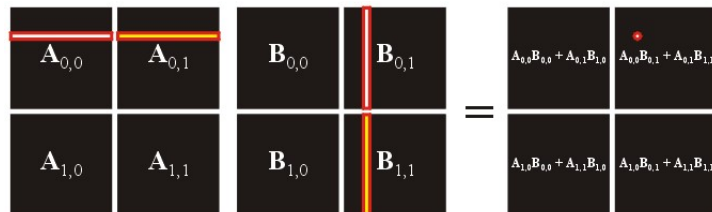
Justification:

c_{ij} is the dot product of the i^{th} row of **A** and the j^{th} column of **B**



Matrix multiplication

This is equivalent for each of the sub-matrices:



Matrix multiplication

We must calculate the four sums-of-products

$$\mathbf{C}_{00} = \mathbf{A}_{00}\mathbf{B}_{00} + \mathbf{A}_{01}\mathbf{B}_{10}$$

$$\mathbf{C}_{01} = \mathbf{A}_{00}\mathbf{B}_{01} + \mathbf{A}_{01}\mathbf{B}_{11}$$

$$\mathbf{C}_{10} = \mathbf{A}_{10}\mathbf{B}_{00} + \mathbf{A}_{11}\mathbf{B}_{10}$$

$$\mathbf{C}_{11} = \mathbf{A}_{10}\mathbf{B}_{01} + \mathbf{A}_{11}\mathbf{B}_{11}$$

This totals 8 products of $(n/2) \times (n/2)$ matrices

- This requires four matrix-matrix additions: $\Theta(n^2)$

Matrix multiplication

The recurrence relation is:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T\left(\frac{n}{2}\right) + \Theta(n^2) & n > 1 \end{cases}$$

Using Master Theorem for Divide and Conquer:

$$T(n) = O(n^{\log_2^8}) = O(n^3) \text{ (not better)}$$

Matrix multiplication

In 1969, Strassen developed a technique for performing matrix-matrix multiplication in $\Theta(n^{\lg(7)}) \approx \Theta(n^{2.807})$ time

- Reduce the number of matrix-matrix products

Matrix multiplication

Consider the following *seven* matrix products

$$\mathbf{M}_1 = (\mathbf{A}_{00} - \mathbf{A}_{10})(\mathbf{B}_{00} + \mathbf{B}_{01})$$

$$\mathbf{M}_2 = (\mathbf{A}_{00} + \mathbf{A}_{11})(\mathbf{B}_{00} + \mathbf{B}_{11})$$

$$\mathbf{M}_3 = (\mathbf{A}_{01} - \mathbf{A}_{11})(\mathbf{B}_{10} + \mathbf{B}_{11})$$

$$\mathbf{M}_4 = \mathbf{A}_{00}(\mathbf{B}_{01} - \mathbf{B}_{11})$$

$$\mathbf{M}_5 = \mathbf{A}_{11}(\mathbf{B}_{10} - \mathbf{B}_{00})$$

$$\mathbf{M}_6 = (\mathbf{A}_{10} + \mathbf{A}_{11})\mathbf{B}_{00}$$

$$\mathbf{M}_7 = (\mathbf{A}_{00} + \mathbf{A}_{01})\mathbf{B}_{11}$$

The four sub-matrices of \mathbf{C} may be written as

$$\mathbf{C}_{00} = \mathbf{M}_3 + \mathbf{M}_2 + \mathbf{M}_5 - \mathbf{M}_7$$

$$\mathbf{C}_{01} = \mathbf{M}_4 + \mathbf{M}_7$$

$$\mathbf{C}_{10} = \mathbf{M}_5 + \mathbf{M}_6$$

$$\mathbf{C}_{11} = \mathbf{M}_2 - \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_6$$

Matrix multiplication

Thus, the new recurrence relation is:

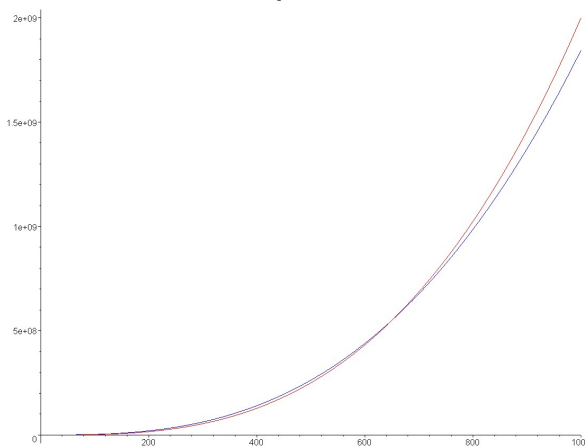
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & n > 1 \end{cases}$$

Using Master Theorem for Divide and Conquer:

$$T(n) = O(n^{\log_2^7}) = O(n^{2.81}) \text{ (better)}$$

Matrix multiplication

Note, however, that there is a lot of additional work required
Counting additions and multiplications:



Matrix multiplication

Examining this plot, and then solving explicitly, we find that Strassen's method only reduces the number of operations for $n > 654$

- Better asymptotic behaviour does not immediately translate into better run-times

The Strassen algorithm is not the fastest

- the Coppersmith–Winograd algorithm runs in $\Theta(n^{2.376})$ time but the coefficients are too large for any problem

Therefore, better asymptotic behaviour does not immediately translate into better run-times

Summary

- Brute Force
 - We consider all possible solutions, and find that solution which is optimal
- Greedy algorithms
 - We extend a partial solution to a feasible solution
- Divide-and-conquer algorithms
 - We divide a problem to many sub-problems
- Next week: Dynamic programming