The Divide and Conquer paradigm

- Basic idea: decomposing a problem into smaller subproblems
- Three steps:
  - Divide problem into subproblems
  - Solve each subproblem (recursively)
  - Combine solution of smaller problems into solution of original problem
- We have already seen several examples of this paradigm:
  - Binary search
  - Quicksort
  - Mergesort
- There is a general method for analyzing many algorithms of this type.

The Divide and Conquer Equation

Suppose that for a problem of size n:

- We divide it into a problems, each of size n/b.
- The dividing and combining requires $f(n)$ work.
- The base case, (e.g., $n = 1$) can be solved with $\Theta(1)$ work.

Then we get the following Divide and Conquer Recurrence Equation:

$$\begin{cases} T(n) &= aT\left(\frac{n}{b}\right) + f(n), \quad \text{if } n > 1 \\ T(1) &= \Theta(1). \end{cases}$$

There are several ways to solve equations of this form (see [GT], section 11.1). We will discuss two of them:

1. A general theorem called the Master Method
2. A Simplified Method that works only when $f(n)$ has the special form $f(n) = n^k$

But first, let's look at how the recursion unfolds ...

Solving a divide-and-conquer recurrence

$$\begin{cases} T(n) &= aT\left(\frac{n}{b}\right) + f(n), \quad \text{if } n > 1 \\ T(1) &= \Theta(1). \end{cases}$$

How many recursive calls do we make?

Level 0 (top level)	1 problem of size n
Level 1	a problems of size $\frac{n}{b}$
Level 2	a^2 problems of size $\frac{n}{b^2}$
...	
Level j	a^j problems of size $\frac{n}{b^j}$

The base case occurs when $\frac{n}{b^j} = 1$. Solve for j: $j = \log_b n$.

For this value of j, the number of problems is $a^j = a^{\log_b n}$.

Rewriting: $a^{\log_b n} = b^{\log_b a \cdot \log_b n} = b^{\log_b n \cdot \log_b a} = n^{\log_b a}$

So the number of recursive calls generated is $\Theta(n^{\log_b a})$.

Solving a divide-and-conquer recurrence

$$\begin{cases} T(n) &= aT\left(\frac{n}{b}\right) + f(n), \quad \text{if } n > 1 \\ T(1) &= \Theta(1). \end{cases}$$

The number of recursive calls generated is $\Theta(n^{\log_b a})$.

This implies:

- If $n^{\log_b a}$ grows significantly faster than $f(n)$, the cost of the recursion dominates and the running time is determined by $n^{\log_b a}$. This is Case 1 in the two theorems that follow.
- If $n^{\log_b a}$ grows significantly slower than $f(n)$, the cost of the non-recursive steps dominates, and the running time is determined by $f(n)$. This is Case 3 in the two theorems that follow.
- If the growth rates of $n^{\log_b a}$ and $f(n)$ are not too far apart, we are between these two cases and something more nuanced happens. This is Case 2 in the two theorems that follow.

A divide-and-conquer theorem

Theorem: Let $a \geq 1$, $b > 1$, and $k \geq 0$ be constants. Let $T(n)$ be defined on the nonnegative integers by the recurrence equation

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^k\right).$$

Then $T(n)$ can be bounded asymptotically as follows:

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

Notes:

1. If $a > b^k$, then $\log_b a > k$, so $n^{\log_b a}$ grows faster than n^k
2. If $a < b^k$, then $\log_b a < k$, so $n^{\log_b a} = o\left(n^k\right)$
3. We interpret $\frac{n}{b}$ to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$.)
4. This is a special case of the "Master Method" (following).
5. We will refer to this theorem as the "Simplified Method"

Some examples using the Simplified Method

Example 1: $T(n) = 9T\left(\frac{n}{3}\right) + n$

Using the Simplified Method:

$$\begin{aligned} a = 9, b = 3, k = 1 \quad &\Rightarrow \quad b^k = 3 \\ &\Rightarrow \quad a > b^k \\ &\Rightarrow \quad \text{Case 1} \end{aligned}$$

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_3 9}\right) = \Theta\left(n^2\right)$$

Some examples using the Simplified Method

Example 2: $T(n) = T\left(\frac{2n}{3}\right) + 1$

Using the Simplified Method:

$$a = 1, b = 3/2, k = 0 \quad \Rightarrow \quad b^k = 1$$
$$\Rightarrow \quad a = b^k$$
$$\Rightarrow \quad \text{Case 2}$$

$$T(n) = \Theta\left(n^k \log n\right) = \Theta\left(n^0 \log n\right) = \Theta\left(\log n\right)$$

Some examples using the Simplified Method

Example 3: $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

Using the Simplified Method:

- Can't use it!!
- RHS is not of the form $\Theta\left(n^k\right)$

The Master Method

Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, [and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,] then $T(n) = \Theta\left(f(n)\right)$.

Notes on the Master Method

Notes:

1. We interpret $\frac{n}{b}$ to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$.
2. The regularity condition ("if $af(n/b) \leq cf(n)$ for...") in Case 3 is a technical condition that only fails in extremely pathological circumstances.
3. The proof of the Master Method is omitted here.
   - A high-level justification is given in the Section 11.1 of [GT].
   - A rigorous proof of a similar theorem is proved in Section 4.4 of [CLRS]. (Note that case (2) in the [CLRS] version is weaker than case (2) of the [GT] version.)

Applying the master method

- If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some $k \geq 0$, then Case 2 applies.
- Otherwise:
  - If $f(n) = O\left(n^{\log_b a}\right)$, then Case 1 is the only case that could apply. (It may or may not apply).
  - If $f(n) = \Omega\left(n^{\log_b a}\right)$, then Case 3 is the only case that could apply. (It may or may not apply).

Some examples using the Master Method

Example 1: $T(n) = 9T\left(\frac{n}{3}\right) + n$

Using the Master Method:

$$a = 9, b = 3, f(n) = n \quad \Rightarrow \quad \log_b a = \log_3 9 = 2$$

$$n^1 = O\left(n^{2-\epsilon}\right) \quad \Rightarrow \quad f(n) = O\left(n^{\log_b a - \epsilon}\right)$$

So Case 1 applies:

$$T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_3 9}\right) = \Theta\left(n^2\right)$$

Some examples using the Master Method

Example 2: $T(n) = T\left(\frac{2n}{3}\right) + 1$

Using the Master Method:

$$a = 1, b = 3/2, f(n) = 1 \quad \Rightarrow \quad \log_b a = \log_{3/2} 1 = 0$$

$$1 = \Theta\left(n^0\right) \quad \Rightarrow \quad f(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_b a} \log^0 n\right)$$

So Case 2 applies with $k = 0$:

$$T(n) = \Theta\left(n^{\log_b a} \log^1 n\right) = \Theta\left(n^{\log_{3/2} 1} \log n\right) = \Theta\left(n^0 \log n\right) = \Theta\left(\log n\right)$$

Some examples using the Master Method

Example 3: $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

Using the Master Method:

$$a = 3, b = 4, f(n) = n \log n \quad \Rightarrow \quad \log_b a = \log_4 3 < 1$$

$$n \log n = \Omega\left(n^{\log_4 3 + \epsilon}\right) \quad \Rightarrow \quad f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$$

So Case 3 applies:

$$T(n) = \Theta(f(n)) = \Theta(n \log n)$$

Some examples using the Master Method

Example 4: $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$

Using the Master Method:

$$a = 2, b = 2, f(n) = n \log n \quad \Rightarrow \quad \log_b a = \log_2 2 = 1$$

$$f(n) = n \log n = \Theta\left(n^{\log_b a} \log^1 n\right)$$

So Case 2 applies with $k = 1$:

$$T(n) = \Theta\left(n^{\log_b a} \log^2 n\right) = \Theta\left(n \log^2 n\right)$$

Note that we cannot use the Simplified Method on this example

Notes on the Master Method vs. the Simplified Method

1. As we have seen, there are cases where the Master Method applies and the Simplified Method does not (e.g., Examples 3 and 4).
2. In the analysis of most of the algorithms in the remainder of this set of notes, we use the Simplified method. In each of these cases, we could have used the Master Method instead.
3. Because there are cases where the Master Method Applies and the Simplified Method does not, you should be familiar with the Master Method as well as the Simplified Method.

Analysis of Binary Search

Top level call is `binarySearch(A,x,0,n-1)`

```
def binarySearch(A,x,first,last):
    if first > last:  return -1
    index = ⌊(first+last)/2⌋
    if x = A[index]:  return index
    else if x < A[index]:  return binarySearch(A,x,first,index-1)
    else:  return binarySearch(A,x,index+1,last)
```

Recurrence equation:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Use Simplified Method: $a = 1$, $b = 2$, $k = 0$ (because $1 = n^0$)

Since $a = b^k$, we apply Case 2:

$$T(n) = O\left(n^k \log n\right) = O\left(n^0 \log n\right) = O(\log n)$$

Analysis of Merge Sort

Top level call is `mergeSort(A,0,n-1)`

```
def mergeSort(A,x,first,last):
    if first < .ast:
        mid = ⌊(first + last)/2⌋;
        mergeSort(A,first,mid)
        mergeSort(A,mid+1,last)
        merge(A,first,mid,mid+1,last)
```

Recurrence equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Use Simplified Method: $a = 2$, $b = 2$, $k = 1$ (because $n = n^1$)

Since $a = b^k$, we apply Case 2:

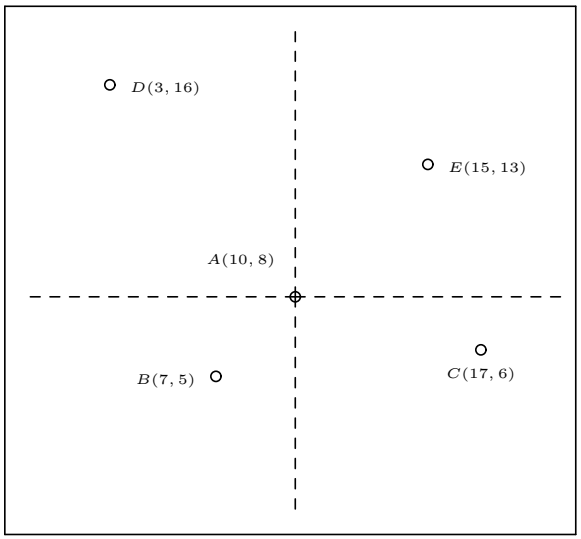
$$T(n) = O\left(n^k \log n\right) = O\left(n^1 \log n\right) = O(n \log n)$$

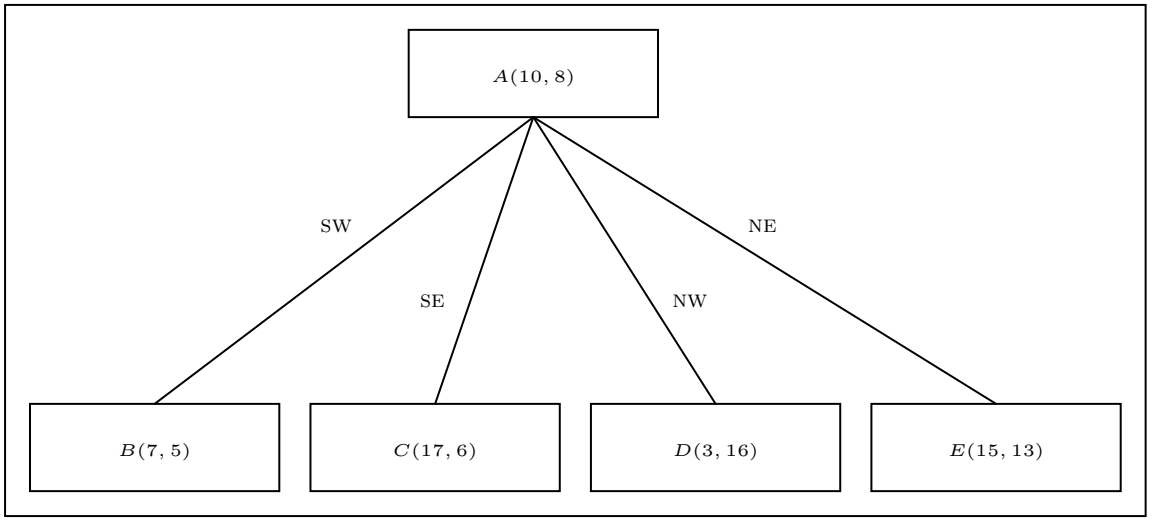
Searching in a point quadtree

A Point Quadtree is a data structure for storing 2-dimensional data, analogous to (1-dimensional) binary search tree.

- Each node stores a point (x, y).
- Each node has 4 child pointers, each of which points to the (possibly empty) root of a subtree. The 4 children are called the southwest (SW), southeast (SE), northwest (NW), and northeast (NE) child.
- If the point stored at a node is (x_0, y_0) then:
  - All points (x, y) stored in the SW subtree have $x < x_0$ and $y < y_0$.
  - All points (x, y) stored in the SE subtree have $x > x_0$ and $y < y_0$.
  - All points (x, y) stored in the NW subtree have $x < x_0$ and $y > y_0$.
  - All points (x, y) stored in the NE subtree have $x > x_0$ and $y > y_0$.

Point Quadtree Example





- A point quadtree is perfect if all non-leaf nodes have 4 nonempty children, and if all leaf nodes have the same depth.
- The point quadtree shown above is perfect, with depth $= 1$.

Exact match in a perfect quadtree with n nodes

Given a pair of values r and s, determine whether the point (r, s) is stored in the tree.

When we visit a node containing (x, y), if $(x, y) \neq (r, s)$, we must (recursively) traverse one subtree of the node. So cost is given by:

$$\begin{cases} T(n) &= T\left(\frac{n}{4}\right) + O(1), \quad \text{if } n > 1 \\ T(1) &= 1 \end{cases}$$

Here: $a = 1$, $b = 4$, $k = 0$, so $a = b^k$. The solution is

$$T(n) = O\left(n^0 \lg n\right) = O(\lg n)$$

Partial match in a perfect quadtree with n nodes

Given a value r, determine whether the quadtree contains some point (r, s) (where s can be arbitrary).

When we visit a node containing (x, y), if $x \neq r$, we must (recursively) traverse two subtrees of the node:

- SW and NW children if $r < x$
- SE and NE children if $r > x$

So cost is given by:

$$\begin{cases} T(n) &= 2T\left(\frac{n}{4}\right) + O(1), \quad \text{if } n > 1 \\ T(1) &= 1 \end{cases}$$

Here: $a = 2$, $b = 4$, $k = 0$, so $a > b^k$. Case 1 applies, and the solution is:

$$T(n) = O\left(n^{\log_b a}\right) = O\left(n^{\log_4 2}\right) = O\left(n^{1/2}\right) = O\left(\sqrt{n}\right)$$

Constructing a binary heap

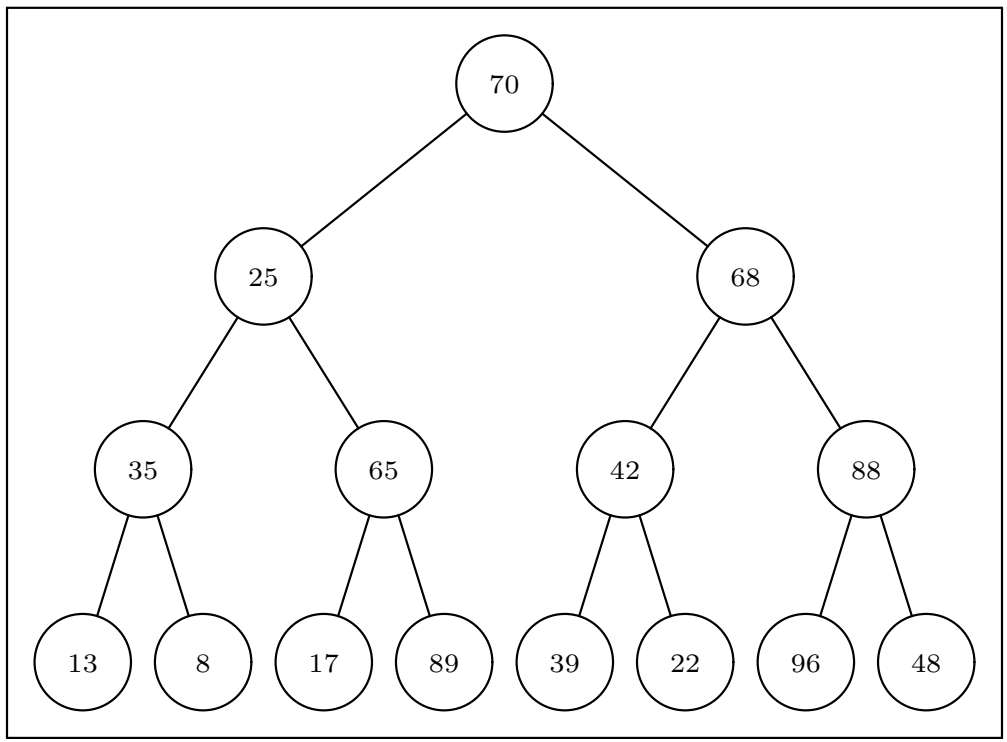
- Earlier we discussed a `heapify()` procedure for constructing a binary heap in linear time.
- Here is a recursive (divide and conquer) approach to the same problem.
  1. Put the data in the heap in arbitrary order. (So H stores the correct data, but does not necessarily satisfy the heap invariant.)
  2. Run the following recursive heapify function. Top-level call is `recHeapify(H,0)`.

```
recHeapify(H,k);
   if (2k+1 <= n) recHeapify(H,2k+1) // Heapify left subtree
   if (2k+2 <= n) recHeapify(H,2k+2) // Heapify right subtree
   siftDown(H,k) // sift down value at index k
```

Recursive Heapify Example

70, 25, 68, 35, 65, 42, 88, 13, 8, 17, 89, 39, 22, 96, 48

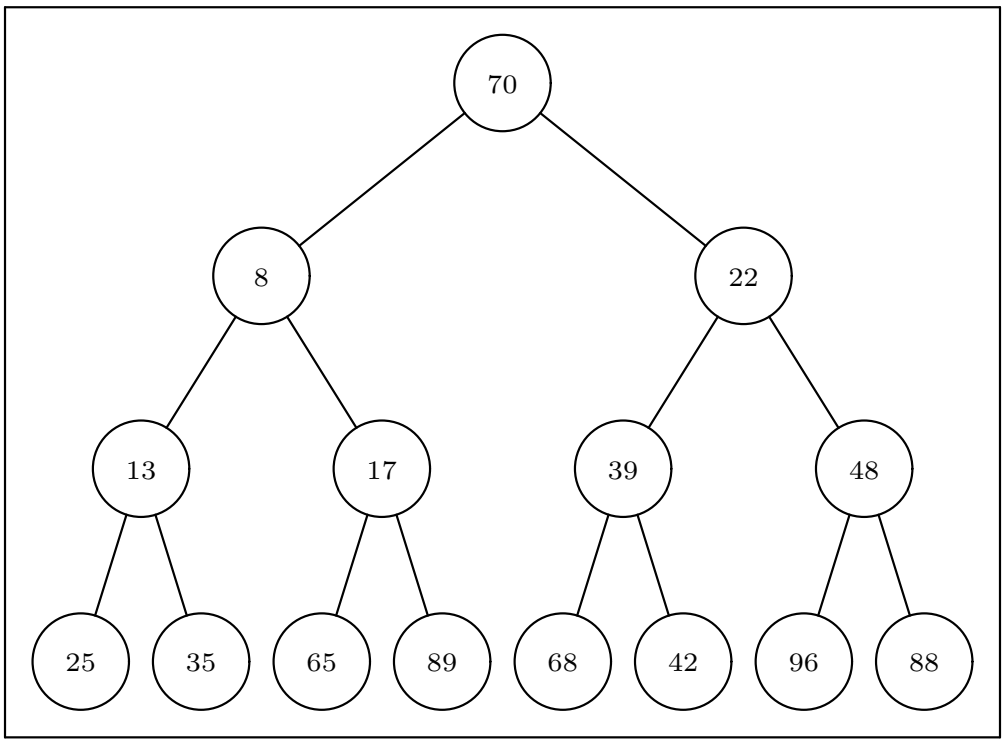
70	25	68	35	65	42	88	13	8	17	89	39	22	96	48
0	1	2	3	4	5	6	7	9	9	10	11	12	13	14



Recursive Heapify Example, continued

After recursive calls to heapify left, right subtree of root:

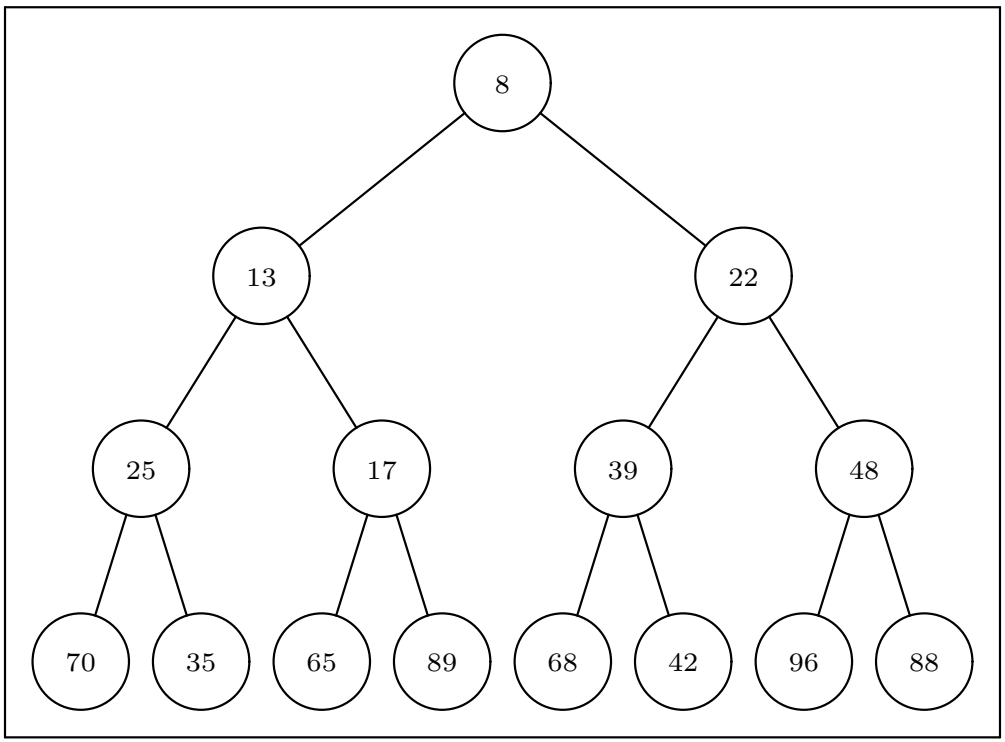
70	8	22	13	17	39	48	25	35	65	89	68	42	96	88
0	1	2	3	4	5	6	7	9	9	10	11	12	13	14



Recursive Heapify Example, continued

After sifting down value at root:

8	13	22	25	17	39	48	70	35	65	89	68	42	96	88
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Analysis of Heap Construction

- Let $T(n)$ be time required to heapify an array of n items.
- Two recursive calls: each one takes time $T(n/2)$.
- Sift-down: $O(\log n)$
- So recurrence equation is

$$T(n) \leq 2T(n/2) + O(\log n)$$

- Solve with Master Method:
  - Since $\log_2 2 = 1$ and $\log n = O(n^{1-\epsilon})$ for some $\epsilon > 0$,

$$T(n) = O(n)$$

1-Dimensional closest pair problem

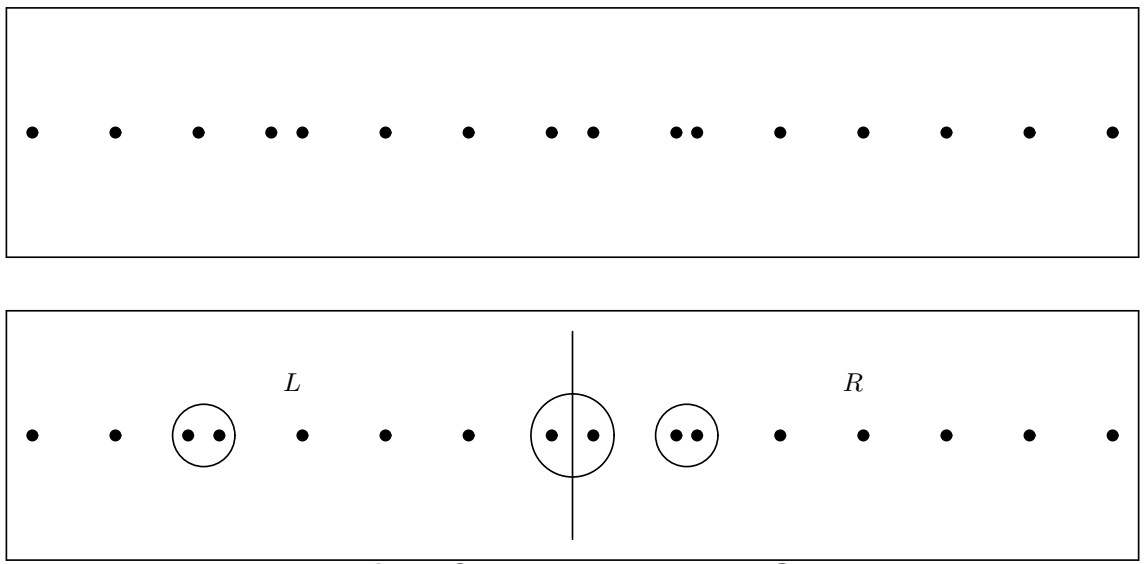
Given a set of n numbers, find the two numbers that are closest together. (The distance between two numbers x and y is $|x - y|$.)

This problem can be solved by sorting. Here we present a divide-and-conquer solution that does not require sorting.

Divide-and-conquer solution of 1-dimensional closest pair problem

If $n \leq 3$, use brute force. Otherwise:

1. Split the set into two sets L and R of equal size, such that all the numbers in L are less than all the numbers in R.
2. Recursively find the closest pair in each of the two sets L and R.
3. Choose the closest pair from 3 candidate pairs:
   - The closest pair in L
   - The closest pair in R
   - The maximum value in L and the minimum value in R



Analysis of Divide-and-conquer solution of 1-dimensional closest pair problem

Cost of the three steps:

1. Splitting into L and R: $O(n)$. (This is because we can find the median of n numbers in O(n) time.)
2. Recursively finding closest pair in sets L and R: $2T(n/2)$
3. Choosing the closest pair from 3 candidate pairs: $O(n)$. (This can actually be done in $O(1)$ time, assuming that max and min were computed in the recursive call in Step 2).

Recurrence equation is:

$$\begin{cases} T(n) &= 2T\left(\frac{n}{2}\right) + O\left(n\right), \quad \text{if } n > 3 \\ T(n) &= O(1), \quad \text{if } n \leq 3 \end{cases}$$

Solution is: $T(n) = O\left(n \lg n\right)$.

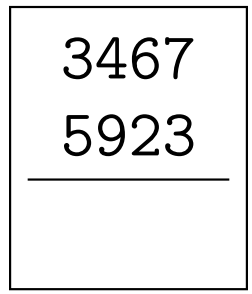
Comparison of divide-and-conquer solution with sorting solution

The divide-and-conquer solution does the same work (asymptotically) as a solution based on sorting.

Advantage: divide-and-conquer solution generalizes to closest-pair problem in two dimensions, sorting solution does not.

Integer Multiplication

Let x, y be two n-bit binary integers (or two n-digit decimal integers). Find their product



- The straightforward algorithm requires $O(n^2)$ operations.
- There is a more efficient approach based on divide-and-conquer.

Divide and Conquer approach to integer multiplication

Let r,s be two n-digit numbers, and let $z = r * s$. Split r and s into two $n/2$ digit numbers:

$$r = r_1 \times 10^{n/2} + r_2 \qquad r \boxed{r_1 \mid r_2}$$

$$s = s_1 \times 10^{n/2} + s_2 \qquad s \boxed{s_1 \mid s_2}$$

We can compute the value of z by:

$$z = (r_1 * s_1) \times 10^n + (r_1 * s_2 + r_2 * s_1) \times 10^{n/2} + (r_2 * s_2)$$

Let $T(n)$ be the time required to multiply two n-digit numbers.

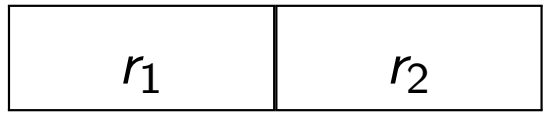
$$T(n) = 4T\left(\frac{n}{2}\right) + O(n).$$

The solution is $T(n) = O(n^2)$.

So we haven't gained anything yet. But ...

Improved algorithm for integer multiplication

$$r = r_1 \times 10^{n/2} + r_2$$



$$s = s_1 \times 10^{n/2} + s_2$$

We can compute the value of z by:

$$\begin{aligned} u &= (r_1 + r_2) * (s_1 + s_2) \\ v &= r_1 * s_1 \\ w &= r_2 * s_2 \\ z &= v \times 10^n + (u - v - w) \times 10^{n/2} + w \end{aligned}$$

This only requires 3 multiplications of two $n/2$-digit numbers. It works because

$$\begin{aligned} u - v - w &= (r_1 + r_2) * (s_1 + s_2) - r_1 * s_1 - r_2 * s_2 \\ &= r_1 * s_2 + r_2 * s_1 . \end{aligned}$$

Improved algorithm for integer multiplication (continued)

Now our recurrence equation is

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n).$$

The solution is

$$T(n) = O\left(n^{\log_2 3}\right) = O\left(n^{1.58496\ldots}\right)$$

This is an improvement over $O\left(n^2\right)$

Example of the two methods

Compute $z = 3467 * 5923$ 3467: | 34 | 67 | 5923: | 59 | 23 |

Straightforward method:

$$\begin{aligned} z &= (34 \times 10^2 + 67) * (59 \times 10^2 + 23) \\ &= (34 * 59) \times 10^4 + (34 * 23 + 67 * 59) \times 10^2 + (67 * 23) \\ &= (2006) \times 10^4 + (782 + 3953) \times 10^2 + 1541 \\ &= (2006) \times 10^4 + (4735) \times 10^2 + 1541 \\ &= 20535041 \end{aligned}$$

Improved method:

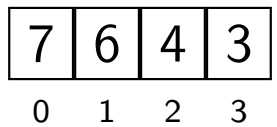
$$\begin{aligned} u &= (34 + 67) * (59 + 23) = 101 * 82 = 8282 \\ v &= 34 * 59 = 2006 \\ w &= 67 * 23 = 1541 \\ z &= v \times 10^4 + (u - v - w) \times 10^2 + w \\ &= 2006 \times 10^4 + (8282 - 2006 - 1541) \times 10^2 + 1541 \\ &= 2006 \times 10^4 + (4735) \times 10^2 + 1541 \\ &= 20535041 \end{aligned}$$

Pseudocode for improved method

```
def Multiply(r,s,n):
  //Split r and s into r1 and r2, s1 and s2
  r1 = r[n/2:n/2]; r2 = r[0:n/2]; s1 = s[n/2:n/2]; s2 = s[0:n/2]
  //Compute u = (r1+r2) * (s1+s2)
  u1 = Add(r1,r2) // u1 = r1+r2
  u2 = Add(s1,s2) // u2 = s1+s2
  u = Multiply(u1,u2,(n/2)+1) // u = u1*u2
  //Compute v=r1*s1, w=r2*s2
  v = Multiply(r1,s1,n/2) // v=r1*s1
  w = Multiply(r2,s2,n/2) // w=r2*s2
  //Compute z = v×10^n+ (u-v-w)×10^(n/2) + w
  u = Subtract(u,v)
  u = Subtract(u,w)
  Shift(u,n/2) // Now u = (u-v-w)×10^(n/2)
  Shift(v,n) // Now v = v×10^n
  z = Add(v,u)
  z = Add(z,w)
  return z
```

Notes on pseudocode

1. Digits are numbered from right to left (i.e., "ones" digit is digit #0, "tens" digit is digit #1, etc.) Digit i is stored in array position i.



2. The functions `add(a,b)` and `subtract(a,b)` return, respectively an array representing $a + b$ and an array representing $a - b$.
3. The operation `shift(a,k)` right-shifts the array a (i.e., left-shifts the number a represents) by k places, filling locations $0, 2, \ldots, k-1$ with zeros. (In essence, it multiplies a by 10^k.)
4. Some care is needed to make sure enough digits are computed. For example, u may have $n/2 + 1$ digits.
5. It is assumed that the temporary arrays (i.e., `u1`, `u2`, `u`, `v`, and `w`) are big enough.
6. The code assumes that the original n is a power of 2. This can be achieved by padding with 0's in the high-order digits. Alternatively, the pseudocode could be modified to be more careful about floors and ceilings.

Matrix multiplication and Strassen's method

Quick review of matrices, matrix multiplication:

- For a matrix A, the element in row i and column j is denoted by $a_{i,j}$.
- If A and B are two $n \times n$ matrices, their product is the matrix $C = A \times B$ defined by

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

  ($A \times B$ is defined as long as the number of columns of A equals the number of rows of B, but we will only worry about square matrices for now.)
- A straightforward computation of $A \times B$ requires $\Theta(n^3)$ operations and exactly n^3 multiplications.
  - Because n^2 entries must be calculated, and each entry requires n multiplications.

Example

$$\begin{bmatrix} 3 & 1 & 4 & 1 & 5 & 2 & 6 & 4 \\ 1 & 6 & 3 & 8 & 5 & 1 & 8 & 4 \\ 2 & 1 & 6 & 5 & 3 & 9 & 7 & 4 \\ 3 & 3 & 1 & 8 & 7 & 5 & 4 & 4 \\ 3 & 1 & 1 & 2 & 1 & 0 & 9 & 1 \\ 1 & 0 & 0 & 2 & 2 & 1 & 2 & 4 \\ 5 & 5 & 6 & 9 & 0 & 3 & 5 & 5 \\ 0 & 0 & 2 & 6 & 8 & 2 & 4 & 5 \end{bmatrix} \times \begin{bmatrix} 1 & 3 & 0 & 0 & 1 & 0 & 6 & 1 \\ 5 & 5 & 8 & 2 & 2 & 5 & 2 & 4 \\ 6 & 5 & 6 & 2 & 9 & 4 & 4 & 1 \\ 2 & 9 & 5 & 2 & 7 & 5 & 7 & 7 \\ 6 & 5 & 5 & 4 & 2 & 5 & 3 & 7 \\ 2 & 1 & 0 & 6 & 9 & 2 & 1 & 4 \\ 6 & 1 & 3 & 0 & 0 & 7 & 2 & 9 \\ 0 & 5 & 9 & 2 & 7 & 6 & 5 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 104 & 96 & 116 & 52 & 104 & 121 & 92 & 135 \\ 145 & 174 & 191 & 68 & 143 & 189 & 138 & 215 \\ 131 & 137 & 141 & 98 & 208 & 160 & 125 & 187 \\ 116 & 165 & 153 & 90 & 161 & 156 & 138 & 197 \\ 78 & 56 & 65 & 14 & 37 & 93 & 64 & 115 \\ 31 & 54 & 62 & 26 & 56 & 60 & 51 & 71 \\ 120 & 184 & 181 & 68 & 194 & 165 & 165 & 176 \\ 100 & 135 & 139 & 70 & 129 & 140 & 109 & 169 \end{bmatrix}$$

$$c_{1,1} = 3 \cdot 1 + 1 \cdot 5 + 4 \cdot 6 + 1 \cdot 2 + 5 \cdot 6 + 2 \cdot 2 + 6 \cdot 6 + 4 \cdot 0 = 104$$

There is a more efficient algorithm for multiplying matrices [Strassen,1969].

Partitioned matrices

We can partition A and B into 4 $\frac{n}{2} \times \frac{n}{2}$ submatrices:

$$A = \left[\begin{array}{c|c} X_{1,1} & X_{1,2} \\ \hline X_{2,1} & X_{2,2} \end{array}\right] \qquad B = \left[\begin{array}{c|c} Y_{1,1} & Y_{1,2} \\ \hline Y_{2,1} & Y_{2,2} \end{array}\right]$$

$$C = \left[\begin{array}{c|c} X_{1,1}Y_{1,1} + X_{1,2}Y_{2,1} & X_{1,1}Y_{1,2} + X_{1,2}Y_{2,2} \\ \hline X_{2,1}Y_{1,1} + X_{2,2}Y_{2,1} & X_{2,1}Y_{1,2} + X_{2,2}Y_{2,2} \end{array}\right]$$

Example of Partitioned matrices

$$\left[\begin{array}{cccc|cccc} 3 & 1 & 4 & 1 & 5 & 2 & 6 & 4 \\ 1 & 6 & 3 & 8 & 5 & 1 & 8 & 4 \\ 2 & 1 & 6 & 5 & 3 & 9 & 7 & 4 \\ 3 & 3 & 1 & 8 & 7 & 5 & 4 & 4 \\ \hline 3 & 1 & 1 & 2 & 1 & 0 & 9 & 1 \\ 1 & 0 & 0 & 2 & 2 & 1 & 2 & 4 \\ 5 & 5 & 6 & 9 & 0 & 3 & 5 & 5 \\ 0 & 0 & 2 & 6 & 8 & 2 & 4 & 5 \end{array}\right] \times \left[\begin{array}{cccc|cccc} 1 & 3 & 0 & 0 & 1 & 0 & 6 & 1 \\ 5 & 5 & 8 & 2 & 2 & 5 & 2 & 4 \\ 6 & 5 & 6 & 2 & 9 & 4 & 4 & 1 \\ 2 & 9 & 5 & 2 & 7 & 5 & 7 & 7 \\ \hline 6 & 5 & 5 & 4 & 2 & 5 & 3 & 7 \\ 2 & 1 & 0 & 6 & 9 & 2 & 1 & 4 \\ 6 & 1 & 3 & 0 & 0 & 7 & 2 & 9 \\ 0 & 5 & 9 & 2 & 7 & 6 & 5 & 5 \end{array}\right]$$

$$= \left[\begin{array}{cccc|cccc} 104 & 96 & 116 & 52 & 104 & 121 & 92 & 135 \\ 145 & 174 & 191 & 68 & 143 & 189 & 138 & 215 \\ 131 & 137 & 141 & 98 & 208 & 160 & 125 & 187 \\ 116 & 165 & 153 & 90 & 161 & 156 & 138 & 197 \\ \hline 78 & 56 & 65 & 14 & 37 & 93 & 64 & 115 \\ 31 & 54 & 62 & 26 & 56 & 60 & 51 & 71 \\ 120 & 184 & 181 & 68 & 194 & 165 & 165 & 176 \\ 100 & 135 & 139 & 70 & 129 & 140 & 109 & 169 \end{array}\right]$$

Example of Partitioned matrices: Computation of upper left quadrant

$$\begin{bmatrix} 3 & 1 & 4 & 1 \\ 1 & 6 & 3 & 8 \\ 2 & 1 & 6 & 5 \\ 3 & 3 & 1 & 8 \end{bmatrix} \times \begin{bmatrix} 1 & 3 & 0 & 0 \\ 5 & 5 & 8 & 2 \\ 6 & 5 & 6 & 2 \\ 2 & 9 & 5 & 2 \end{bmatrix} + \begin{bmatrix} 5 & 2 & 6 & 4 \\ 5 & 1 & 8 & 4 \\ 3 & 9 & 7 & 4 \\ 7 & 5 & 4 & 4 \end{bmatrix} \times \begin{bmatrix} 6 & 5 & 5 & 4 \\ 2 & 1 & 0 & 6 \\ 6 & 1 & 3 & 0 \\ 0 & 5 & 9 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 34 & 43 & 37 & 12 \\ 65 & 120 & 106 & 34 \\ 53 & 86 & 69 & 24 \\ 40 & 101 & 70 & 24 \end{bmatrix} + \begin{bmatrix} 70 & 53 & 79 & 40 \\ 80 & 54 & 85 & 34 \\ 78 & 51 & 72 & 74 \\ 76 & 64 & 83 & 66 \end{bmatrix}$$

$$= \begin{bmatrix} 104 & 96 & 116 & 52 \\ 145 & 174 & 191 & 68 \\ 131 & 137 & 141 & 98 \\ 116 & 165 & 153 & 90 \end{bmatrix}$$

Divide-and-conquer matrix multiplication

This gives us a way of using divide-and-conquer to perform matrix multiplication.

To multiply two $n \times n$ matrices

- Partition each of the matrices into 4 $\frac{n}{2} \times \frac{n}{2}$ submatrices
- Perform a 2×2 matrix multiplications
  - Each element of each of the 2×2 matrices is a $\frac{n}{2} \times \frac{n}{2}$ submatrix.
  - Since multiplying two 2×2 matrices requires 8 multiplications, there are a total of 8 recursive calls.

$$C = \left[\begin{array}{c|c} X_{1,1}Y_{1,1} + X_{1,2}Y_{2,1} & X_{1,1}Y_{1,2} + X_{1,2}Y_{2,2} \\ \hline X_{2,1}Y_{1,1} + X_{2,2}Y_{2,1} & X_{2,1}Y_{1,2} + X_{2,2}Y_{2,2} \end{array}\right]$$

Analysis of Divide-and-conquer matrix multiplication

Let $T(n)$ be the cost of multiplying two $n \times n$ matrices.
Performing this multiplication requires:

- 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ submatrices. [Cost $= 8T\left(\frac{n}{2}\right)$]
- 4 additions of $\frac{n}{2} \times \frac{n}{2}$ submatrices. [Cost $= O\left(n^2\right)$]

Recurrence equation:

$$T(n) = 8T\left(\frac{n}{2}\right) + O\left(n^2\right).$$

Here $a = 8, b = 2, k = 2$, so

$$T(n) = O\left(n^{\log_2 8}\right) = O\left(n^3\right).$$

and we haven't gained anything yet.

But we can incorporate a better way of multiplying 2×2 matrices...

Strassen's improved scheme for multiplying 2×2 matrices

Strassen[1969] showed how to multiply two 2×2 matrices using 7 multiplications and 18 additions.

Let X, Y be 2×2 matrices, $Z = X \times Y$. To compute Z:

$$
\begin{aligned}
r_1 &= (x_{11} + x_{22}) * (y_{11} + y_{22}) \\
r_2 &= (x_{21} + x_{22}) * y_{11} \\
r_3 &= x_{11} * (y_{12} - y_{22}) \\
r_4 &= x_{22} * (y_{21} - y_{11}) \\
r_5 &= (x_{11} + x_{12}) * y_{22} \\
r_6 &= (x_{21} - x_{11}) * (y_{11} + y_{12}) \\
r_7 &= (x_{12} - x_{22}) * (y_{21} + y_{22})
\end{aligned}
$$

$$
\begin{aligned}
z_{11} &= r_1 + r_4 - r_5 + r_7 \\
z_{21} &= r_2 + r_4 \\
z_{12} &= r_3 + r_5 \\
z_{22} &= r_1 + r_3 - r_2 + r_6
\end{aligned}
$$

Example of Strassen's 2×2 multiplication scheme

$$X = \begin{bmatrix} 3 & 1 \\ 2 & 6 \end{bmatrix} \qquad Y = \begin{bmatrix} 5 & 7 \\ 4 & 8 \end{bmatrix} \qquad X \times Y = \begin{bmatrix} 19 & 29 \\ 34 & 62 \end{bmatrix}$$

$$\begin{aligned}
r_1 &= (x_{11} + x_{22}) * (y_{11} + y_{22}) = (3+6) * (5+8) = 9 + 13 = 117 \\
r_2 &= (x_{21} + x_{22}) * y_{11} = (2+6) * 5 = 8 * 5 = 40 \\
r_3 &= x_{11} * (y_{12} - y_{22}) = 3 * (7-8) = 3 * (-1) = -3 \\
r_4 &= x_{22} * (y_{21} - y_{11}) = 6 * (4-5) = 6 * (-1) = -6 \\
r_5 &= (x_{11} + x_{12}) * y_{22} = (3+1) * 8 = 4 * 8 = 32 \\
r_6 &= (x_{21} - x_{11}) * (y_{11} + y_{12}) = (2-3) * (5+7) = (-1) * 12 = -12 \\
r_7 &= (x_{12} - x_{22}) * (y_{21} + y_{22}) = (1-6) * (4+8) = (-5) * 12 = -60
\end{aligned}$$

$$\begin{aligned}
z_{11} &= r_1 + r_4 - r_5 + r_7 = 117 + (-6) - 32 + (-60) = 19 \\
z_{21} &= r_2 + r_4 = 40 + (-6) = 34 \\
z_{12} &= r_3 + r_5 = (-3) + 32 = 29 \\
z_{22} &= r_1 + r_3 - r_2 + r_6 = 117 + (-3) - 40 + (-12) = 62
\end{aligned}$$

Analysis of Strassen's method for divide-and-conquer matrix multiplication

Let $T(n)$ be the cost of multiplying two $n \times n$ matrices.Performing this multiplication requires:

- 7 multiplications of $\frac{n}{2} \times \frac{n}{2}$ submatrices. [Cost $= 7T\left(\frac{n}{2}\right)$]
- 18 additions/subtractions of $\frac{n}{2} \times \frac{n}{2}$ submatrices. [Cost $= O\left(n^2\right)$]

Recurrence equation:

$$T(n) = 7T\left(\frac{n}{2}\right) + O\left(n^2\right).$$

Here $a = 7, b = 2, k = 2$, so

$$T(n) = O\left(n^{\log_2 7}\right) = O\left(n^{\log_2 7}\right) = O\left(n^{2.807\ldots}\right).$$

The improvement comes from eliminating one recursive call (one multiplication)