# Divide and Conquer

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

# The **Divide-and-Conquer** Approach

- **Divide** the given problem into *a number of subproblems* that are *smaller instances of the same problem*.
- **Conquer** the subproblems by solving them *recursively*.
  - Or, use brute-force if a subproblem is small enough.
- **Combine** the solutions for the subproblems to obtain the solution for the original problem.

# If you prefer some pseudocode…

**Solve(I):**

```
if (I is small enough)
  solution = DirectSolve(I)
else
  <I_1,I_2,…,I_k> = DivideProblem(I)
  for (j=1 to k)
    solution_j = Solve(I_j)
  solution = Combine(solution_1,…,solution_k)
return solution
```

Or, use brute-force if (sub)problem is simple.

Divide the problem into smaller subproblems.

**Recursively** solve subproblems.

Combine solutions of subproblems to get solution for original problem.

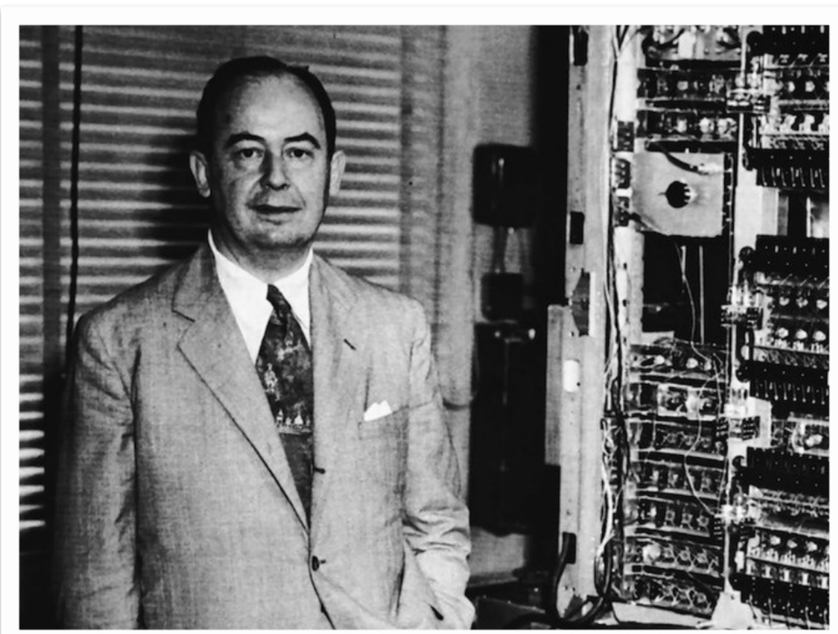
# Correctne
# Conquer

```
Solve(I):
if (I is small enough)
  solution = DirectSolve(I)
else
  <I_1,I_2,...,I_k> = DivideProblem(I)
  for (j=1 to k)
    solution_j = Solve(I_j)
  solution = Combine(solution_1,...,solution_k)
return solution
```

- How to prove t
  alg?
  - Use induction (of course...)
- **Induction basis:** prove the algorithm can correctly solve small problem instances.
  - Prove DirectSolve is correct if $|I| \leq c$.
- **Induction hypothesis:** the algorithm can correctly solve any problem instance of size at most, say, $n$.
  - Solve is correct if $|I| \leq n$.
- **Inductive step:** assuming induction hypothesis, prove the algorithm can correctly solve problem instance of size $n + 1$.
  - Assume Solve is correct if $|I| \leq n$.

# MergeSort

- An efficient divide-and-conquer algorithm for sorting.
- Invented by *John von Neumann* in the 1940s.



***John von Neumann***
***Dec 1903 – Feb 1957***

*Hungarian-American, mathematician, physicist, computer scientist, and polymath.*

# MergeSort

**<u>Solve(I):</u>** **Divide-and-Conquer Template**

if (I is small enough)
  solution = DirectSolve(I)
else
  $<I_1, I_2, \ldots, I_k>$ = DivideProblem(I)
  for (j=1 to k)
    $\text{solution}_j = \text{Solve}(I_j)$
  $\text{solution} = \text{Combine}(\text{solution}_1, \ldots, \text{solution}_k)$
return solution

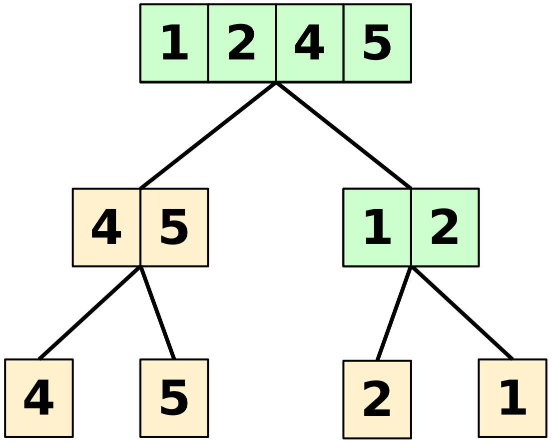
Take a size $m$ sorted array and a size $m'$ sorted array, return a sorted array of size $m + m'$ in $O(m + m')$ time.

**<u>MergeSort(A[1…n]):</u>** **The MergeSort Algorithm**

```
if (n==1)
  sol[1…n] = A[1…n]
else
  solLeft[1…(n/2)] = MergeSort(A[1…(n/2)])
  solRright[1…(n/2)] = MergeSort(A[(n/2+1)…n])
  sol[1…n] = Merge(solLeft[1…(n/2)],solRight[1…(n/2)])
return sol[1…n]
```

# Sample execution of **MergeSort**

```
MergeSort(A[1…n]):
if (n==1)
  sol[1…n] = A[1…n]
else
  solLeft[1…(n/2)] = MergeSort(A[1…(n/2)])
  solRright[1…(n/2)] = MergeSort(A[(n/2+1)…n])
  sol[1…n] = Merge(solLeft[1…(n/2)],solRight[1…(n/2)])
return sol[1…n]
```



# Correctness of **MergeSort**

```
MergeSort(A[1…n]):
if (n==1)
  sol[1…n] = A[1…n]
else
  solLeft[1…(n/2)] = MergeSort(A[1…(n/2)])
  solRright[1…(n/2)] = MergeSort(A[(n/2+1)…n])
  sol[1…n] = Merge(solLeft[1…(n/2)],solRight[1…(n/2)])
return sol[1…n]
```

Take a size $m$ sorted array and a size $m'$ sorted array, return a sorted array of size $m + m'$ in $O(m + m')$ time.

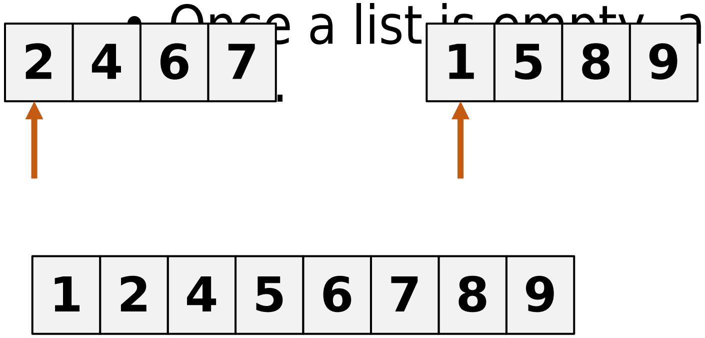
**Induction basis:** MergeSort is correct when $n = 1$.

**Induction hypothesis:** Assume MergeSort is correct if $n \leq n'$.

**Inductive step:** MergeSort is correct when $n = n' + 1$.

# The **Merge** Subroutine

- Merge two sorted lists $L$ and $R$ into a sorted whole $S$.
  - Scan $L$ and $R$ from left to right,
    let $l_i$ and $r_j$ be current elements in $L$ and $R$ respectively.
  - If $l_i \leq r_j$ then append $l_i$ to $S$ and advance $l_i$ (i.e., increase $i$).
  - If $l_i > r_j$ then append $r_j$ to $S$ and advance $r_j$ (i.e., increase $j$).
  - Once a list is empty, append the remaining of the other list .




**Correctness of this routine?**

Find proper loop invariant,
and apply induction…

**Runtime of this routine?**

On input of size $m$ and $m'$,
runtime is $O(m + m')$.

# Time complexity of **MergeSort**

```
MergeSort(A[1…n]):
if (n==1)
  sol[1…n] = A[1…n]
else
  solLeft[1…(n/2)] = MergeSort(A[1…(n/2)])
  solRright[1…(n/2)] = MergeSort(A[(n/2+1)…n])
  sol[1…n] = Merge(solLeft[1…(n/2)],solRight[1…(n/2)])
return sol[1…n]
```

- Let $T(n)$ be the runtime of MergeSort on instance of size $n$.
- Clearly, $T(1) = c_1 = \Theta(1)$ for some constant $c_1$.
- For larger $n$, $T(n) = 2 \cdot T(n/2) + c_2 \cdot n = 2T(n/2) + \Theta(n$

# Time complexity of **MergeSort**

A **recurrence** equation:

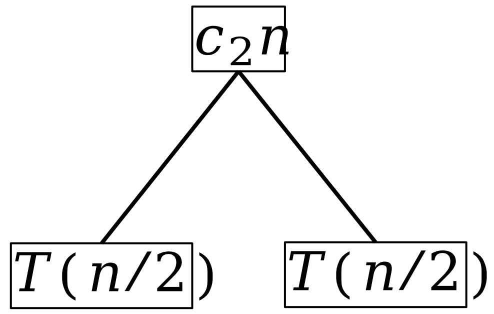
$\boxed{T(n)}$

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

# Time complexity of **MergeSort**

A **recurrence** equation:

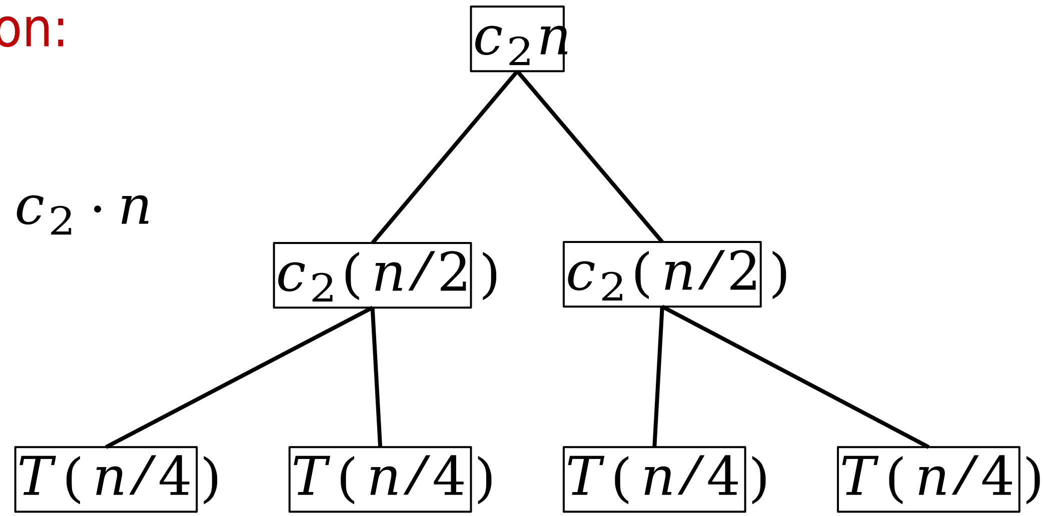
$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$



# Time complexity of **MergeSort**

A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

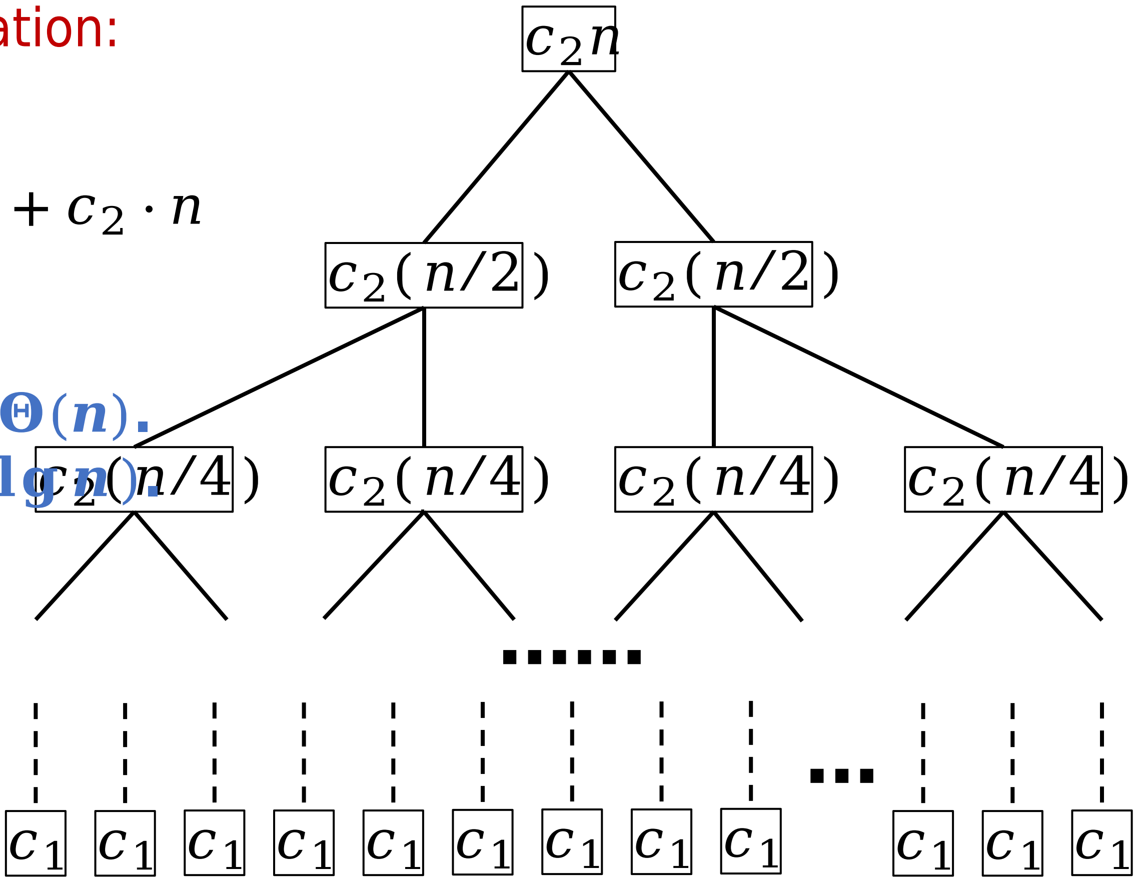


# Time complexity of **MergeSort**

A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

**$\lg n + 1$ levels.**
**Each level incur $\Theta(n)$.**
**Total cost is $\Theta(n \lg n)$.**




**Recursion tree**.

# Iterative **MergeSort**

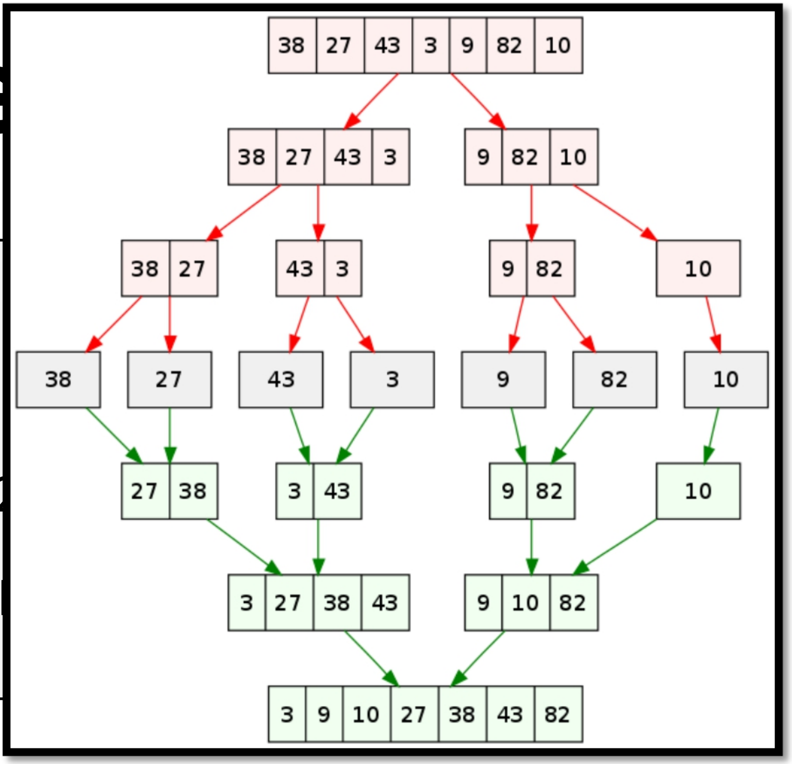
```
MergeSort(A[1...n]):
if (n==1)
  sol[1...n] = A[1...n]
else
  solLeft[1...(n/2)] = MergeSort(A[1...(n/2)])
  solRright[1...(n/2)] = MergeSort(A[(n/2+1)...n])
  sol[1...n] = Merge(solLeft[1...(n/2)],solRight[1...(n/2)])
return sol[1...n]
```

Any recursive algorithm can be converted into an iterative on
we just simulate the **call stack**!

# Iterative **Merge**

**<u>MergeSort(A[1…n]):</u>**

```
if (n==1)
  sol[1…n] = A[1…n]
else
  solLeft[1…(n/2)] = MergeSort(A[1…(n/2
  solRright[1…(n/2)] = MergeSort(A[(n/2
  sol[1…n] = Merge(solLeft[1…(n/2)],sol
return sol[1…n]
```




**<u>MergeSortIter(A[1…n]):</u>**

```
Deque Q
for (i=1 to n)
  Q.AddLast(A[i])
while (Q.Size()>1)
  L=Q.RemoveFirst(),R=Q.RemoveFirst()
  Q.AddLast(Merge(L,R))
return Q.RemoveFirst()
```

Do “merge” operation layer by layer!

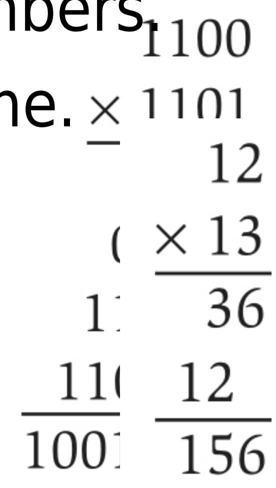
Time complexity is again $\Theta(n \lg n)$.

# Integer Multiplication

- We all know how to do this, since primary school.
- This method can be extended to binary numbers.

**How fast is this method?**

- Consider multiplying two $n$ digits binary numbers.
- Assume single-digit operations takes unit time.
  - Such as addition, multiplication.
- Total time complexity is $O(n^2)$.
- Can we do better, with divide-and-conquer?



# Integer Multiplication

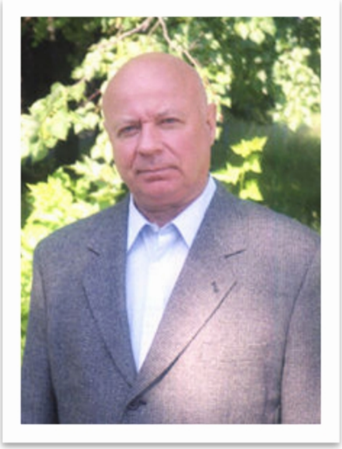
- Assume we want to multiply $x$ and $y$, each having $n$ bits.
- Split each of $x$ and $y$ into their left and right halves.
  - $x = 2^{n/2} \cdot x_L + x_R$ and $y = 2^{n/2} \cdot y_L + y_R$
- $xy = 2^n \cdot x_L y_L + 2^{n/2} \cdot (x_L y_R + x_R y_L) + x_R y_R$
  - Only need four multiplications, instead of six.
- Apply above strategy recursively until $n = 1$
- Recurrence: $T(n) = 4 \cdot T(n/2) + O(n)$

- Time complexity is $T(n) = O(n^2)$, *we are not doing better*!

# Integer Multiplication

- Can we do better than $O(n^2)$?
  - Even great minds once thought we can't.
  - E.g., Andrey Kolmogorov conjectured "no" in 196

- Yet, we can!
  - Anatoly Karatsuba, then a 23-year-old student, found an algorithm in one week!



Karatsuba's algorithm for

# Integer Multiplication

**FastMulti(x, y):**

```
if (x and y are both of 1 bit)
  return x*y
xl, xr = most, least significant |x|/2 bits of x
yl, yr = most, least significant |y|/2 bits of y
z1 = FastMulti(xl,yl)
z2 = FastMulti(xr,yr)
z3 = FastMulti(xl+xr,yl+yr)
return z1*(2^n)+(z3-z1-z2)*(2^(n/2))+z2
```

$x_R y_R$

- We only need **three** multiplications, instead of **four**!
- $T(n) = 3 \cdot T(n/2) + O(n)$
- $T(n) = O(n^{\lg 3}) = O(n^{1.59})$

# Integer Multiplication

- The story didn’t end there…
  - Andrei Toom and Stephen Cook generalizes Karatsuba’s idea:
    $O(n^{1+1/(\lg k)})$ for any $k$, and the constant in $O(\cdot)$ depends on $k$.
  - Arnold Schönhage and Volker Strassen uses FFT:
    $O(n \cdot \log n \cdot \log \log n)$
  - Finally, in March 2019, David Harvey and Joris van der

**Integer multiplication in time $O(n \log n)$**

David Harvey and Joris van der Hoeven

Abstract. We present an algorithm that computes the product of two $n$-bit integers in $O(n \log n)$ bit operations.

Can we do better?

# Matrix Multiplication

- Suppose we want to multiply two $n \times n$ matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$.
- The most straightforward method needs $\Theta(n^3)$ time.

- Matrix multiplication can be performed *block-wise*!
- $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}$ and $\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix}$
- $\boldsymbol{XY} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix} = \begin{bmatrix} \boldsymbol{AE} + \boldsymbol{BG} & \boldsymbol{AF} + \boldsymbol{BH} \\ \boldsymbol{CE} + \boldsymbol{DG} & \boldsymbol{CF} + \boldsymbol{DH} \end{bmatrix}$

- Recurrence: $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$
- $T(n) = \Theta(n^3)$, we are not doing better…

Strassen's algorithm for

# Matrix Multiplication

- Multiply two $n \times n$ matrices $\boldsymbol{X} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}$ and $\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{E} & \boldsymbol{F} \\ \boldsymbol{G} & \boldsymbol{H} \end{bmatrix}$
- $\boldsymbol{XY} = \begin{bmatrix} \boldsymbol{P}_5 + \boldsymbol{P}_4 - \boldsymbol{P}_2 + \boldsymbol{P}_6 & \boldsymbol{P}_1 + \boldsymbol{P}_2 \\ \boldsymbol{P}_3 + \boldsymbol{P}_4 & \boldsymbol{P}_1 + \boldsymbol{P}_5 - \boldsymbol{P}_3 - \boldsymbol{P}_7 \end{bmatrix}$
- $\boldsymbol{P}_1 = \boldsymbol{A}(\boldsymbol{F} - \boldsymbol{H})$, $\boldsymbol{P}_2 = (\boldsymbol{A} + \boldsymbol{B})\boldsymbol{H}$, $\boldsymbol{P}_3 = (\boldsymbol{C} + \boldsymbol{D})\boldsymbol{E}$, $\boldsymbol{P}_4 = \boldsymbol{D}(\boldsymbol{G} - \boldsymbol{E})$
- $\boldsymbol{P}_5 = (\boldsymbol{A} + \boldsymbol{D})(\boldsymbol{E} + \boldsymbol{H})$, $\boldsymbol{P}_6 = (\boldsymbol{B} - \boldsymbol{D})(\boldsymbol{G} + \boldsymbol{H})$, $\boldsymbol{P}_7 = (\boldsymbol{A} - \boldsymbol{C})(\boldsymbol{E} + \boldsymbol{F})$
- Recurrence: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$

# Substitution method (or, guess and verify)

- The substitution method:
  - Guess the form of the solution;
  - Use induction to find proper constants and prove the solution works.
- Recurrence: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$
- $T(n) = 7 \cdot T(n/2) + cn^2$, $T(1) = c$
- Let’s guess $T(n) \le d \cdot n^{\lg 7} = O(n^{\lg 7})$
- **Induction basis:**
  - $T(1) = c \le d \cdot 1^{\lg 7}$, so long as $d \ge c$
- **Inductive step:**
  - $T(n) = 7 \cdot T(n/2) + cn^2 \le 7d \cdot (n/2)^{\lg 7} + cn^2 = dn^{\lg 7} + cn^2$

Time complexity of Strassen's algorithm

# Substitution method (or, guess and verify)

- $T(n) = 7 \cdot T(n/2) + cn^2$, $T(1) = c$
- Guess $T(n) \le d \cdot n^{\lg 7} = O(n^{\lg 7})$ does not work out...
- $O(n^{\lg 7})$ is in fact the right answer...
  - So we add some lower order term (such as $n^2$) to our guess?
  - No, we should *subtract* some lower order term from our guess!
  - Subtraction gives us ***stronger induction hypothesis*** to work with!

Time complexity of Strassen's algorithm

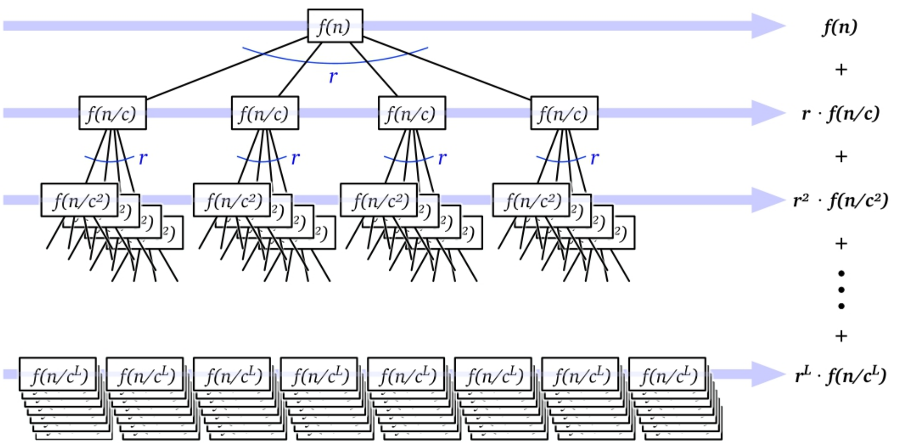
# Substitution method (or, guess and verify)

- $T(n) = 7 \cdot T(n/2) + cn^2$, $T(1) = c$
- ~~Guess~~ $\cancel{T(n) \le d \cdot n^{\lg 7} = O(n^{\lg 7})}$ ~~does not work~~ out...
- Guess $T(n) \le dn^{\lg 7} - d'n^2 = O(n^{\lg 7})$
- **Induction basis:**
  - $T(1) = c \le d \cdot 1^{\lg 7} - d' \cdot 1^2$, so long as $d - d' \ge c$
- **Inductive step:**
  - $T(n) = 7 \cdot T(n/2) + cn^2 \le 7d \cdot (n/2)^{\lg 7} - 7d' \cdot (n/2)^2 + cn^2$
    $= dn^{\lg 7} - (7d'/4 - c)n^2 \le dn^{\lg 7} - d'n^2$, so long as $3d'/4 \ge c$

Solving recurrence

# The recurrence-tree method

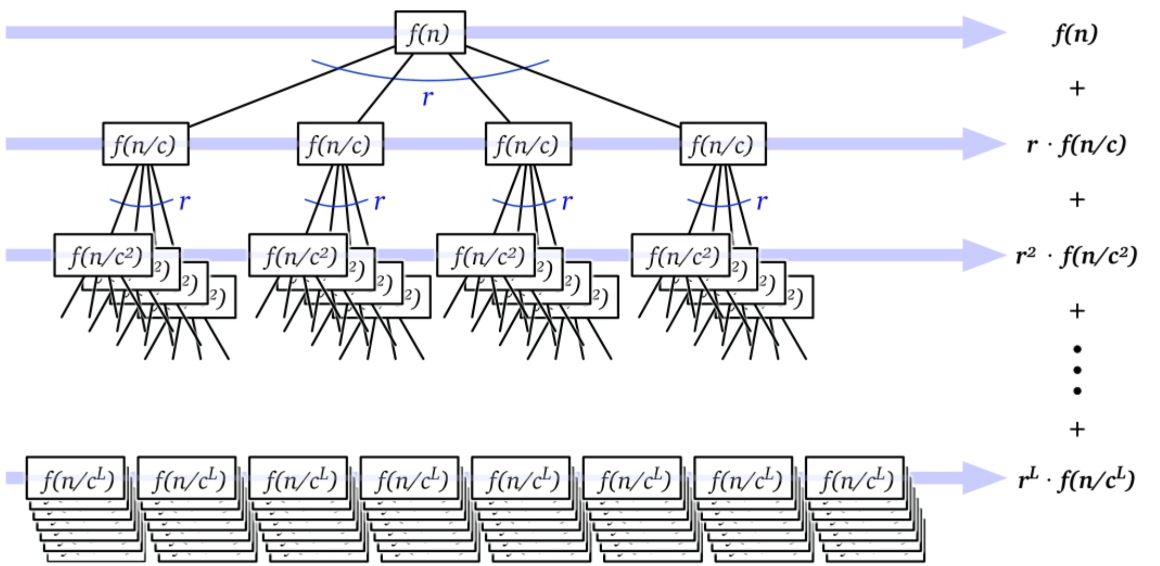
- A great tool for solving divide-and-conquer recurrences.
    - Simple, pictorial, yet general.
- A **recursion tree** is a rooted tree with one node for each recursive subproblem.
- The **value of each node** is the time spent on that subproblem *excluding* recursive calls
- The **sum of all values** i
runtime of the algorithm.



Solving recurrence

# The recurrence-tree method

- Typical divide-and-conquer approach:
  - Divide a size $n$ problem into $r$ subproblems each of size $n/c$, the cost for "divide" and "combine" is $f(n)$.
  - Solve problem directly if $n \leq n_0$ for some small constant $n_0$.
- $\cancel{T(n) = r \cdot T(n/c) + f(n), T(n_0)} = c_0$
- $T(n) = r \cdot T(n/c) + f(n), T(1) = f(1)$



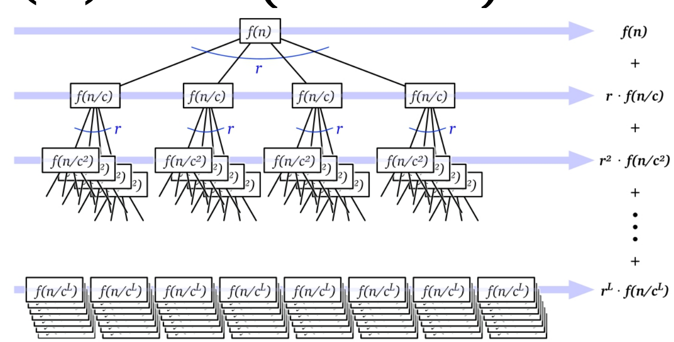
Solving recurrence

# The recurrence-tree method

- $T(n) = r \cdot T(n/c) + f(n)$, $T(1) = f(1)$
- Total cost is $\sum_{i=0}^{L} r^i \cdot f(n/c^i)$, where $L = \log_c n$

Three common cases for the series:

- **Decreasing** (exponentially): $T(n) = O\big(f(n)\big)$
  - Cost dominated by top level, such as $T(n) = T(n/2) + n$
- **Equal**: $T(n) = O(f(n) \cdot L) = O(f(n) \cdot \lg n)$
  - All levels have equal cost, such as $T(n) = 2T(n/2) + n$
- **Increasing** (exponentially): $T(n) = O\big(n^{\log_c r}\big)$
  - Cost dominated by bottom level, such as $T(n) = 4T(n/2) + n$
  - $T(n) = O\big(r^{\log_c n}\big) = O\big(n^{\log_c r}\big)$



The recurrence-tree method

# Master 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

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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

- The Master Theorem does not cover all cases!
- Be careful what does, e.g., $f(n) = O(n^{\log_b a - \epsilon})$, mean.
  - If $a = b = 2$ and $f(n) = n/\lg n$, case one does *not* apply!

Solving recurrence

# The recurrence-tree method

- $T(n) = r \cdot T(n/c) + f(n)$, $T(n_0) = c_0$

- What if $n/c$ is not an integer?
  - In MergeSort, $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$
  - Omitting ceiling/flooring usually does not affect asymptotic results.
- What if a subproblem's size is not "perfect"?
  - In FastMulti, $T(n) \le 3 \cdot T(\lceil n/2 \rceil + 1) + \Theta(n)$
  - Often they do not affect asymptotic results.
- What if subproblems are of different sizes?
  - E.g., $T(n) = T(n/3) + T(2n/3) + \Theta(n)$
  - Recurrence-tree can often give good intuition,<br>then use substitution method to obtain the final result.

The divide-and-conquer approach

# Summary

- **Divide**, **Conquer** (recursively or directly), and **Combine**.
- Same problem can be divided in different ways, leading to different algorithms with different performances!
  - MergeSort uses half-and-half split, how about 1-and-(n-1) split?
  - Another splitting method leads to QuickSort. (We'll learn it later…)
- Correctness of divide-and-conquer algorithms:
  - Use mathematical induction (of course…)
- Time complexity of divide-and-conquer algorithms:
  - Recursion-tree method (master theorem), substitution method

# Reading

- [CLRS] Ch.2 (2.3), Ch.4
- [Erickson v1] Ch.1 (excluding 1.5 and 1.8)