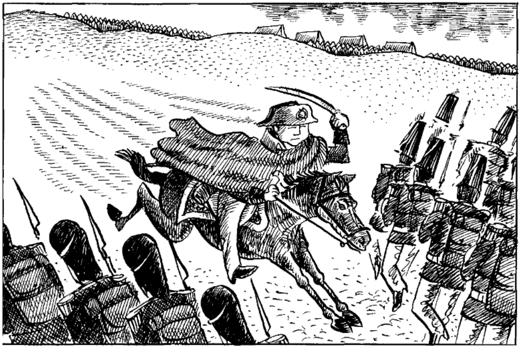
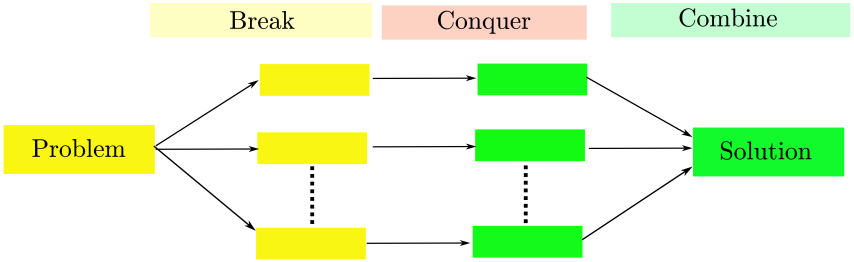
# Chapter 2

# Divide & Conquer

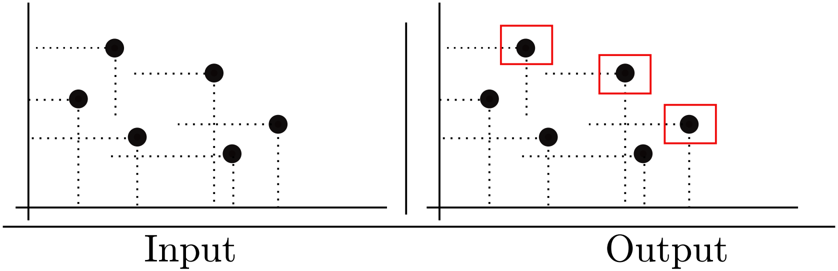


- Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these subproblems into an overall solution.



- In many settings where divide and conquer is applied (as we will see), the brute-force algorithm may already be polynomial time, and the divide and conquer strategy is serving to reduce the running time to a lower polynomial.

## The Maxima-Set Problem



**Input description:** a set, $S$, of $n$ points in the plane.
**Problem description:** A point is a maximum point in $S$ if there is no other point, $(x', y')$, in $S$ such that $x \leq x'$ and $y \leq y'$. The problem is to find a subset of $S$ including all maximum points. This subset is called the maxima set.

**Motivation:**
This problem is motivated from multi-objective optimization, where we are interested in optimizing choices that depend on multiple variables. For instance, purchasing a cell-phone with the best possible specification with the lowest possible price.

- Points that are not members of the maxima set can be eliminated from consideration, since they are dominated by another point in $S$. Thus, finding the maxima set of points can act as a kind of filter that selects out only those points that should be candidates for optimal choices.

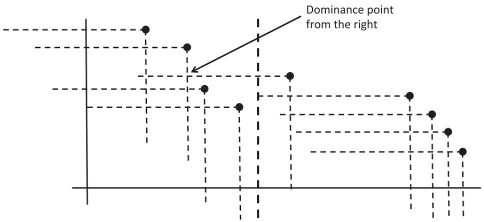
$Stop\&Think$

there is a simple divide-and-conquer algorithm for constructing the maxima set of points in $S$. But what is that?

**Solution:**

1. Order $S$ lexicographically.
   that is, we order . primarily on $x$-coordinates and then by $y$-coordinates if there are ties.
2. Divide $S$ using a vertical line through the median point, $p$.
3. Next, we recursively solve the maxima-set problem for the set of points on the left of this line and also for the points on the right.
4. The combination step ....

## The Combination Step:



Given the left and right solutions,

- the maxima set of points on the right are also maxima points for $S$.
- But some of the maxima points for the left set might be dominated by a point from the right, namely the point, $q$, that is leftmost.
- So then we do a scan of the left set of maxima, removing any points that are dominated by $q$, until reaching the point where $q$'s dominance extends.
- The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for $S$.

```
**Algorithm MaximaSet(S)**
Input: A set, $S$, of $n$ points in the plane
Output: The set, $M$ , of maxima points in $S$
if $n \leq 1$ then
    return $S$
Let $p$ be the median point in $S$, by lexicographic $(x, y)$-coordinates
Let $L$ be the set of points lexicographically less than $p$ in $S$
Let $G$ be the set of points lexicographically greater than or equal to p in S
$M_1 \leftarrow MaximaSet(L)$
$M_2 \leftarrow MaximaSet(G)$
Let $q$ be the lexicographically smallest point in $M_2$
for each point, $r$, in $M_1$ do
    if $x(r) \leq x(q)$ and $y(r) \leq y(q)$ then
        Remove $r$ from $M_1$
return $M_1 \cup M_2$
```

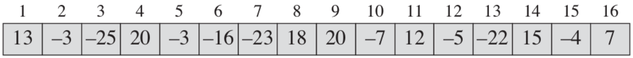
For the running time of the divide-and-conquer maxima-set algorithm, we can write

$$T(n) \leq \begin{cases} 0 & if\ n < 2 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & if\ n \geq 2 \end{cases}$$

$\Rightarrow T(n) \in O(n \log(n))$ why?

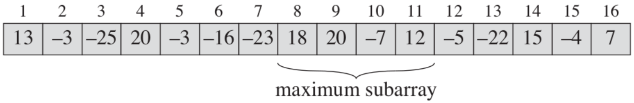
## Maximum Subarray

Input



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | –3 | –25 | 20 | –3 | –16 | –23 | 18 | 20 | –7 | 12 | –5 | –22 | 15 | –4 | 7 |

Output

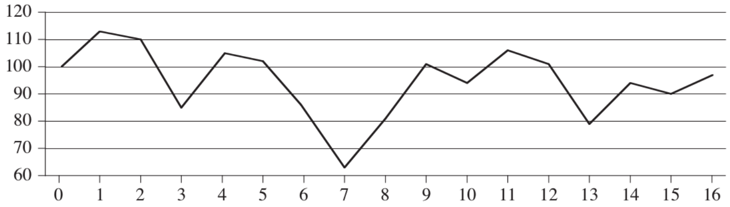


| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 13 | –3 | –25 | 20 | –3 | –16 | –23 | 18 | 20 | –7 | 12 | –5 | –22 | 15 | –4 | 7 |

**Input description:** An array, $A$, of $n$ numbers.
**Problem description:** We want to find the nonempty, contiguous subarray of $A$ whose values have the largest sum. We call this contiguous subarray the maximum subarray. For example, in the array above, the maximum subarray of $A[1 \cdots 16]$ is $A[8 \cdots 11]$, with the sum 43.

## Motivation

Stock Market: You are allowed to buy one unit of stock only one time and then sell it at a later date, buying and selling after the close of trading for the day. You are allowed to learn what the price of the stock will be in the future. Your goal is to maximize your profit.



| Day | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| Change | | 13 | −3 | −25 | 20 | −3 | −16 | −23 | 18 | 20 | −7 | 12 | −5 | −22 | 15 | −4 | 7 |

## In Class Question 3

What is the time complexity of the brute-force (exhaustive search) algorithm?

- The maximum-subarray problem is interesting only when the array contains some negative numbers. If all the array entries were non-negative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

**Brute-force solution:**

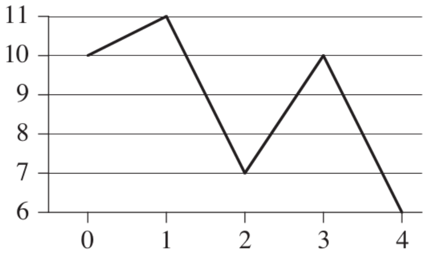
just try every possible pair of buy and sell dates: $\binom{n}{2}$, the best we can hope for is to evaluate each pair of dates in constant time $\Longrightarrow$ this approach would take $\Omega(n^2)$

*Stop&Think*

There is a divide-and-conquer algorithm for maximizing the profit with $O(n \log n)$ complexity. But what is that?

Is this a true strategy? Buying at the lowest price or selling at the highest price.

A counter-example:

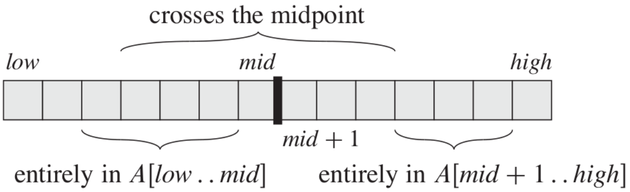


**Solution:** Suppose we want to find a maximum subarray of the subarray $A[low \cdots high]$.

1. Divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say $mid$, of the subarray, and consider the subarrays $A^L = A[low \cdots mid]$ and $A^R = A[mid + 1 \cdots high]$.
2. Assume that the problem is solved for the two smaller subarrays. That is the maximum subarray of each has been found.
3. The combination step ...

There are three possibilities:

- The solution to $A^L$ is also the solution for $A$.
- The solution to $A^R$ is also the solution for $A$.
- The maximum subarray of the entire array is not completely in $A^L$ nor in $A^R$. But part of the maximum subarray is in $A^L$, and part of that is in $A^R$.


crosses the midpoint
$low$
$mid$
$high$
$mid + 1$
entirely in $A[low \mathinner{..} mid]$
entirely in $A[mid + 1 \mathinner{..} high]$

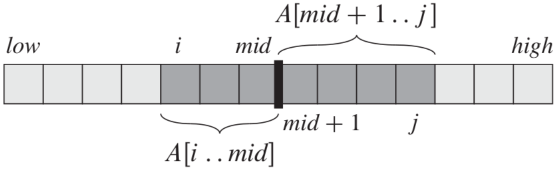
Maximum subarray of $A$: take the subarray with the largest sum of the above three candidates.

سه احتمال وجود دارد:
راه حل AL نیز راه حل A است.
راه حل AR نیز راه حل A است.
حداکثر زیرآرایه کل آرایه نه به طور کامل در AL و نه در AR است. اما بخشی از حداکثر زیرآرایه در AL و بخشی از آن در AR است.

حداکثر زیرآرایه A: زیرآرایه ای را با بیشترین مجموع سه کاندید بالا در نظر بگیرید

حالا میایم سراغ divied and conquer برای این مسئله :
کاری که باید صورت بگیره اینه که ما یه نقطه میانی رو در نظر می گیریم برای این دنباله
حالا اون جوابی که دنبالش می گردیم از سه حالت خارج نیست:
یا کلا سمت چپه یا کلا سمت راسته یا نصفش سمت چپه و نصفش سمت راسته
پس وقتی این دنباله رو شکوندیم باید بیایم ماکزیمم جمع زیر دنباله سمت چپی رو به دست بیاریم و ماکزیمم جمع زیر دنباله سمت راستی هم همین طور و ماکزیمم جمعی هم که از نقطه میانی رد می شه هم به دست بیاریم و ببینیم کدومش بهتره
پس T(n) ما می شه:
هزینه بهترین از نقطه میانیT(n/2)+T(n/2)+

- We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray $A[low \cdots high]$
- This problem is not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint.



```
FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
$left\text{-}sum = -\infty$
$sum = 0$
**for** $i = mid$ **downto** $low$
    $sum = sum + A[i]$
    **if** $sum > left\text{-}sum$
        $left\text{-}sum = sum$
        $max\text{-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
**for** $j = mid + 1$ **to** $high$
    $sum = sum + A[j]$
    **if** $sum > right\text{-}sum$
        $right\text{-}sum = sum$
        $max\text{-}right = j$
**return** $(max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum)$
```

Complexity of the FIND-MAX-CROSSING- SUBARRAY: If the subarray $A[low \cdots high]$ contains $n$ entries (so that $n = high - low + 1$), we claim that the call FIND-MAX-CROSSING- SUBARRAY( $A, low, mid, high$ ) takes $\Theta(n)$ time. Why?...
(► hint: count the number of loops' iterations)

```
FIND-MAXIMUM-SUBARRAY$(A, low, high)$
 1  if $high == low$
 2      return $(low, high, A[low])$          // base case: only one element
 3  else $mid = \lfloor (low + high)/2 \rfloor$
Conquer Recursively
 4      $(left\text{-}low, left\text{-}high, left\text{-}sum) =$
            FIND-MAXIMUM-SUBARRAY$(A, low, mid)$
 5      $(right\text{-}low, right\text{-}high, right\text{-}sum) =$
            FIND-MAXIMUM-SUBARRAY$(A, mid + 1, high)$
Combine
 6      $(cross\text{-}low, cross\text{-}high, cross\text{-}sum) =$
            FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
 7      if $left\text{-}sum \geq right\text{-}sum$ and $left\text{-}sum \geq cross\text{-}sum$
 8          return $(left\text{-}low, left\text{-}high, left\text{-}sum)$
 9      elseif $right\text{-}sum \geq left\text{-}sum$ and $right\text{-}sum \geq cross\text{-}sum$
10          return $(right\text{-}low, right\text{-}high, right\text{-}sum)$
11      else return $(cross\text{-}low, cross\text{-}high, cross\text{-}sum)$
```

► Because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.

**Time Complexity Analysis:**

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + f(n), \quad f(n) \in \Theta(n)$$

$\Longrightarrow T(n) \in \Theta(n \log n)$ why?

►We see that the divide-and-conquer method yields an algorithm that is asymptotically faster than the brute-force method. There is in fact a linear-time algorithm for the maximum-subarray problem, and it does not use divide- and-conquer.

## Solving the Recurrence

## Example (1)

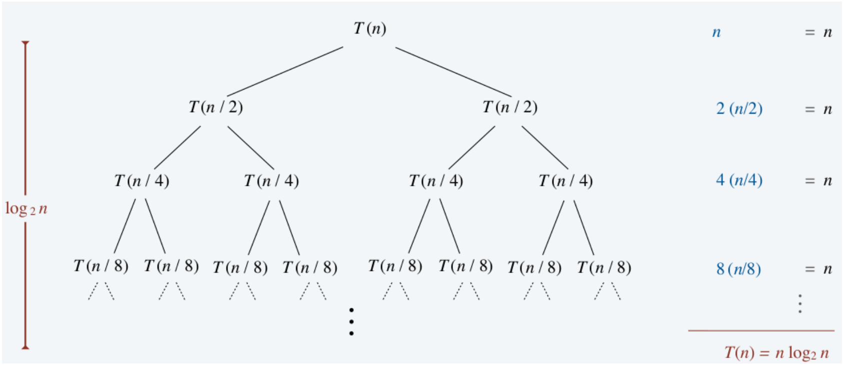
$$T(n) = \begin{cases} 0 & if\ n < 2 \\ T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & if\ n \geq 2 \end{cases}$$

First method: Guess and Prove by induction Initially we assume $n$ is a power of 2 and replace $\leq$ with $=$ in the recurrence.

## Theorem

*If $T(n)$ satisfies the following recurrence, then $T(n) = n \log_2 n$.*

$$T(n) = \begin{cases} 0 & if\ n < 2 \\ 2T(n/2) + n & if\ n \geq 2 \end{cases}$$


$T(n)$
$T(n/2)$
$T(n/2)$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$T(n/4)$
$T(n/8)$
$\log_2 n$
$n \quad = n$
$2\,(n/2) \quad = n$
$4\,(n/4) \quad = n$
$8\,(n/8) \quad = n$
$T(n) = n \log_2 n$

* $\sum_{i=1}^{n} i = n(n+1)/2$
* $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
* $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$
* $\sum_{i=0}^{n} r^i = \left(r^{n+1}-1\right)/(r-1)$

## Proof by induction.

Induction base: when $n = 1, T(1) = 0 = n \log_2 n$.
Inductive hypothesis: assume $T(n) = n \log_2 n$
Induction step:
$T(2n) = 2T(n) + 2n = 2n \log_2 n + 2n = 2n(\log_2(n) + 1) = 2n \log_2(2n)$.
□

Theorem

*If $T(n)$ satisfies the following recurrence, then $T(n) \leq n\lceil \log_2 n \rceil$.*

$$T(n) \leq \begin{cases} 0 & if\ n < 2 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & if\ n \geq 2 \end{cases}$$

## Proof by induction.

Induction base: $n = 1$.
Inductive hypothesis: assume true for $1, 2, \cdots, n-1$.
Induction step:
Define $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$ and note that $n = n_1 + n_2$.

$$\begin{aligned} T(n) &\leq T(n_1) + T(n_2) + n \leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &\leq n_1 \lceil \log_2 n_2 \rceil + n_2 \lceil \log_2 n_2 \rceil + n = n \lceil \log_2 n_2 \rceil + n \\ &\leq n(\lceil \log_2 n \rceil - 1) + n \quad \text{(for the reason, see the note below)} \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

□

note: $n_2 = \lceil n/2 \rceil \leq \lceil 2^{\lceil \log_2 n \rceil}/2 \rceil = 2^{\lceil \log_2 n \rceil}/2 \rightarrow \log_2 n_2 \leq \lceil \log_2 n \rceil - 1$

### Example (2)

$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$

If we guess the asymptomatic behavior of $T(n)$ as $T(n) = O(n \lg n)$, then

Assume by induction that this bound holds for all numbers at least as big as $n_0$ and less than $n$. Therefore if $n \geq 2n_0$, then

$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$

Substituting this into recurrence, yields

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq 2(cn/2 \lg(n/2)) + n \\ &= cn \lg(n/2) + n \\ &= cn \lg(n) - cn + n \\ &\leq cn \lg(n) \end{aligned}$$

for $c \geq 1$.

**Avoiding pitfalls**: Avoid using asymptotic notation in the inductive hypothesis for the substitution method because it's error prone. Wrong assumption: $T(n) = O(n)$

$$\begin{aligned} T(n) &\leq 2O(\lfloor n/2 \rfloor) + n \\ &= 2O(n/2) + n \\ &= O(n) \end{aligned}$$

The problem with this reasoning is that the constant hidden by the $O$-notation changes.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \nleq cn \end{aligned}$$

## Second Method: Use the Master Theorem

**Theorem**

*If $T(n)$ satisfies the following recurrence, then $T(n) \in O(n \log_2 n)$.*

$$T(n) \leq \begin{cases} 0 & if\ n < 2 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & if\ n \geq 2 \end{cases}$$

**Proof.**

Because the Master Theorem Says! □

## Master Theorem: (Mastering Recursions)

- Recall that a divide-and-conquer algorithm solves a problem by dividing its given instance into several smaller instances, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.
- Assuming that all smaller instances have the same size $n/b$, with $a$ of them being actually solved, we get the following recurrence valid

$$T(n) = aT(n/b) + g(n) \tag{1}$$

where $a \geq 0$, $b \geq 1$, and $g(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and combining their solutions. Recurrence (1) is called the **general divide-and-conquer recurrence**.

- In equation (1), we interpret n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$.
- Applying backward substitutions to (1), assuming $n = b^k$:

$$\begin{aligned}
T(n) = T(b^k) &= aT(b^{k-1}) + g(b^k) \\
&= a[aT(b^{k-2}) + g(b^{k-1})] + g(b^k) \\
&= a^2T(b^{k-2}) + ag(b^{k-1}) + g(b^k) \\
&= a^2[aT(b^{k-3}) + g(b^{k-2})] + ag(b^{k-1}) + g(b^k) \\
&= a^3T(b^{k-3}) + a^2g(b^{k-2}) + ag(b^{k-1}) + g(b^k) \\
&= \cdots \\
&= a^kT(1) + a^{k-1}g(b^1) + a^{k-2}g(b^2) + \cdots + a^0g(b^k) \\
&= a^k \left( T(1) + \sum_{j=1}^{k} g(b^j)/a^j \right)
\end{aligned}$$

since $a^k = a^{\log_b n} = n^{\log_b a}$

$$T(n) = n^{\log_b a} \left( T(1) + \sum_{j=1}^{\log_b n} g(b^j)/a^j \right) \qquad (2)$$

Obviously, the order of growth of solution $T(n)$ depends on the values of the constants $a$ and $b$ and the order of growth of the function $g(n)$. Under certain assumptions about $g(n)$ discussed in the next slides, we can simplify formula (2) and get explicit results about the order of growth of $T(n)$.

## Theorem (Master Theorem–A simplified version)

*Let $T(n)$ be an eventually nondecreasing function that satisfies the recurrence*

$$T(n) = aT(n/b) + g(n) \quad for\ n > 1$$

*where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, and $a \geq 0, b > 1$. If $g(n) \in \Theta(n^d)$ where $d \geq 0$, then*

$$T(n) = \begin{cases} \Theta(n^d) & if\ \log_b(a) < d \\ \Theta(n^d \log n) & if\ \log_b(a) = d \\ \Theta(n^{\log_b a}) & if\ \log_b(a) > d \end{cases}$$

*First presented by J. Bentley, D. Haken, and J. B. Saxe in 1980, described as a "unifying method". The name "master theorem" was popularized by the widely used textbook "Introduction to Algorithms" by CLRS.–Wikipedia*

**Example**

$T(n) = 8T(n/4) + 5n^2$ for $n > 1$, $n$ a power of 4
T(1)=3

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

We have $\log_4 8 < 2$, and therefore, $T(n) \in \Theta(n^d) = \Theta(n^2)$

**Example**

$T(n) = 9T(n/3) + 5n^1$ for $n > 1$, $n$ a power of 3
T(1)=7

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

We have $\log_3 9 > 1$, and therefore, $T(n) \in \Theta(n^{\log_3 9}) = \Theta(n^2)$

**Definition:** Let $f(n)$ be a nonnegative function defined on the set of natural numbers. $f(n)$ is called smooth if it is eventually nondecreasing and

$$f(bn) \in \Theta(f(n))$$

for any fixed integer $b \geq 2$.

- It is easy to check that functions which do not grow too fast, including $\log n$, $n$, $n \log n$, and $n^\alpha$ where $\alpha \geq 0$, are smooth.
    - Example: $f(n) = n \log n$ is smooth because $f(2n) = 2n \log 2n = 2n(\log 2 + \log n) = (2 \log 2)n + 2n \log n \in \Theta(n \log n)$.
- Fast-growing functions, such as $a^n$ where $a > 1$ and $n!$, are not smooth.
    - Example: $f(n) = 2^n$ is not smooth because $f(2n) = 2^{2n} = 4^n \notin \Theta(2^n)$.

**Lemma: Smoothness Rule**
Let T (n) be an eventually nondecreasing function and f (n) be a smooth function. If

$$T(n) \in \Theta(f(n)) \quad \text{for values of } n \text{ that are powers of } b,$$

where $b \geq 2$, then

$$T(n) \in \Theta(f(n))$$

(The same is true for $O$, $\Omega$ and $o$.)

Proof is left as an exercise!

- We can investigate the time efficiency of recursive algorithms first for $n$'s that are powers of $b$ (Most often $b = 2$, but it can be any integer $\geq$ 2). Then, we can use this Lemma to extend the results to all values of $n$.

Geometric series

$$\sum_{k=n_1}^{n_2} r^k = \frac{r^{n_1} - r^{n_2+1}}{1-r}$$

**Proof of the Master Theorem:**
We will prove the theorem for the principal special case of $g(n) = n^d$.
Moreover, first we consider the case where $n = b^k$. Based on (2):
$T(n) = n^{\log_b a}\left(T(1) + \sum_{j=1}^{\log_b n}(b^d/a)^j\right)$
Using geometric series formula, we have

$$\sum_{j=1}^{\log_b n}(b^d/a)^j = \begin{cases}(b^d/a)\frac{(b^d/a)^{\log_b n}-1}{(b^d/a)-1} & if\ b^d \neq a\\ \log_b n & if\ b^d = a\end{cases}$$

As a result,

$$\begin{cases}\sum_{j=1}^{\log_b n}(b^d/a)^j \in \Theta((b^d/a)^{\log_b n}) & if\ b^d/a > 1\\ \sum_{j=1}^{\log_b n}(b^d/a)^j \in \Theta(1) & if\ b^d/a < 1\\ \sum_{j=1}^{\log_b n}(b^d/a)^j = \log_b n & if\ b^d/a = 1\end{cases}$$

Therefore, if $b^d/a > 1$

$$\begin{aligned} T(n) \in n^{\log_b a}\Theta((b^d/a)^{\log_b n}) &= \Theta(n^{\log_b a}(b^d/a)^{\log_b n}) \\ &= \Theta(a^{\log_b n}(b^d/a)^{\log_b n}) \\ &= \Theta((b^d)^{\log_b n}) \\ &= \Theta(n^{\log_b b^d}) = \Theta(n^d) \end{aligned}$$

if $b^d/a < 1$

$$T(n) \in \Theta(n^{\log_b a})$$

and if $b^d/a = 1$

$$T(n) = n^{\log_b a}[T(1) + \log_b n] \in \Theta(n^{\log_b a}\log_b n) = \Theta(n^d \log_b n)$$

Accordingly, proof is completed by noting to Lemma.

$\square$

## Theorem (Master Theorem)

*Let $T(n)$ be an eventually nondecreasing function that satisfies the recurrence*

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

*where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, and $a \geq 0, b > 1$. Then**

$$T(n) = \begin{cases} \Theta(f(n)) & \textit{if } \exists \epsilon > 0, f(n) = \Omega(n^{\log_b a + \epsilon}) \\ \Theta(n^{\log_b a} \log^{k+1} n) & \textit{if } \exists k \geq 0, f(n) = \Theta(n^{\log_b a} \log^k n) \\ \Theta(n^{\log_b a}) & \textit{if } \exists \epsilon > 0, f(n) = O(n^{\log_b a - \epsilon}) \end{cases}$$

* For the first case, $f(n)$ must additionally be a smooth function.

### Example

$$T(n) = 2T(n/2) + n \lg n$$

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

$$n^{\log_2 2} = n \rightarrow T(n) = \Theta(n \lg^2 n)$$

## In Class Question 4

Solve with master: $T(n) = 2T(n/2) + n/\lg n$

Example (Solving recurrences with a change of variables)

$T(n) = 2T(\sqrt{n}) + \Theta(\lg n)$

$m = \lg n \to T(2^m) = 2T(2^{m/2}) + \Theta(m)$
$\to S(m) = 2S(m/2) + \Theta(m), S(m) = T(2^m)$
$\to S(m) = \Theta(m \lg m)$
$\to S(\lg n) = T(n) = \Theta(\lg n \lg \lg n)$

- As we have discussed, floors and ceilings don't change the asymptotic behavior of the solution if $f(n)$ is smooth.

$$T(\lfloor n/b \rfloor) \to T(n/b)$$
$$T(\lceil n/b \rceil) \to T(n/b)$$

- Under the same condition, we can also ignore constant additive terms

$$T(\lfloor n/b + c \rfloor) \to T(n/b)$$
$$T(\lceil n/b + c \rceil) \to T(n/b)$$

Example

$T(n) = 2T(\lfloor n/2 + 17 \rfloor) + n$

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

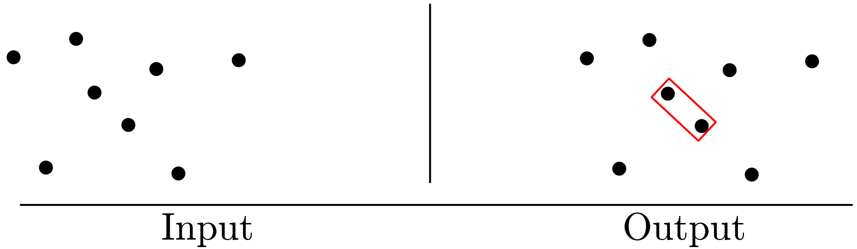
$T(n) = O(n \lg n)$

**Final Points:**

• When we state and solve recurrences, we often omit floors, ceilings, constant additive term, and boundary conditions.

• Occasionally, we shall see recurrences that are not equalities but rather inequalities, such as $T(n) \leq 2T(n/2) + \Theta(n)$. Because such a recurrence states only an upper bound on $T(n)$, we will couch its solution using $O$-notation rather than, $\Theta$-notation. Similarly, if the inequality were reversed to $T(n) \geq 2T(n/2) + \Theta(n)$ then because the recurrence gives only a lower bound on $T(n)$, we would use $\Omega$-notation in its solution.

## Finding the closest pair of points



**Input description:** a set $P$ of $n \geq 2$ points. We assume that the points are distinct.
**Problem description:** finding the closest pair of points in $P$. "Closest" refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

**Motivation:**
The problem was first considered by M. I. Shamos and D. Hoey in the early 1970s, as part of their project that founded the field of **computational geometry**.
Many applications:

- Air or sea traffic-control systems might need to identify the two closest vehicles in order to detect potential collisions.
- Cluster analysis in statistics based on some similarity metric. A bottom-up algorithm begins with each element as a separate cluster and merges them into successively larger clusters by combining the closest pair of clusters.

**Brute-force solution:**
just measure the distance between any two points in $P \Longrightarrow$ this approach would take $\Omega(n^2)$

$\boxed{Stop\&Think}$

There is a divide-and-conquer algorithm with $O(n \log n)$ complexity. But what is that?

• Assume that at the beginning we sort the point in $P$ by both the x-coordinate and y-coordinate and put the results in the $X$ and $Y$ lists, respectively.

**Solution:**

1. Divide:
    - Divide the array $X$ into arrays $X_L$ and $X_R$ , which contain $X[1...\lceil n/2 \rceil]$ and $X[\lceil n/2 \rceil + 1...n]$.
    - Similarly, divide the array $Y$ into arrays $Y_L$ and $Y_R$ , which contain the points of $X_L$ and $X_R$ respectively.
2. Conquer: make two recursive calls, one to find the closest pair of points in $X_L$ and the other to find the closest pair of points in $X_R$ . The inputs to the first call are arrays $X_L$ and $Y_L$ ; the second call receives the inputs $X_R$ , and $Y_R$. Let the closest-pair distances returned for $X_L$ and $X_R$ be $\delta_L$ and $\delta_R$ , respectively, and let $\delta = \min(\delta_L, \delta_R)$.
3. The combination step ...

$\boxed{\text{In order to reach to } O(n \log n) \text{ time complexity, we have to accomplish both the division and combination steps in } O(n).}$

Before delving into the "combination" step, let's verify whether or not we can "divide" in $O(n)$.

## Division in $O(n)$:

- Making $X_L$ and $X_R$ out of $X$ in $O(n)$ is straight-forward.

$$X_L = X[1 \cdots \lceil n/2 \rceil], \quad X_R = X[\lceil n/2 \rceil + 1 \cdots n]$$

- But how to make $Y_L$ and $Y_R$ out of $Y$ which, respectively, include the the points in $X_L$ and $X_R$ but sorted based on the $y$-coordinate?

This is the trick:

```
let $Y_L[1 \,..\, Y.length]$ and $Y_R[1 \,..\, Y.length]$ be new arrays
$Y_L.length = Y_R.length = 0$
for $i = 1$ to $Y.length$
    if $Y[i] \in X_L$
        $Y_L.length = Y_L.length + 1$
        $Y_L[Y_L.length] = Y[i]$
    else $Y_R.length = Y_R.length + 1$
        $Y_R[Y_R.length] = Y[i]$
```

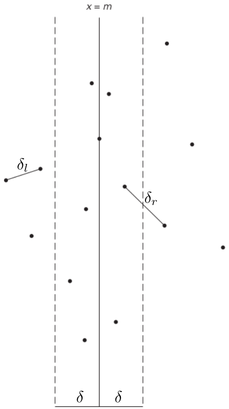
Therefore, we are able to divide in $O(n)$. Now let's proceed to find out how we can combine in just $O(n)$.

This is The Funniest Part ☺

## Combination in O(n)

✓ **Fact 1:** The closest pair is either the pair with distance $\delta$ found by one of the recursive calls, or it is a pair of points with one point in $X_L$ and the other in $X_R$.

✓ **Fact 2:** If there exists $p_l \in X_L$ and $p_r \in X_R$ for which $d(p_l, p_r) < \delta$, then each of $p_l$ and $p_r$ lies within a distance $\delta$ of line $l$ (the vertical line passing through the rightmost point in $X_L$, denoted by $x = m$).



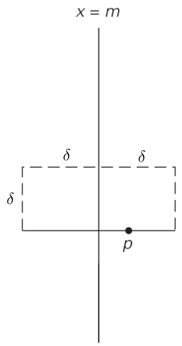
Proof:

$$d(p_l, p_r) < \delta \rightarrow$$

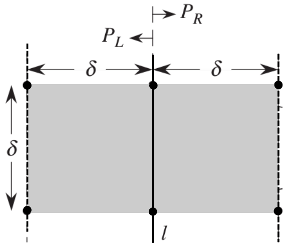
$$p_r.x - p_l.x \leq \delta \rightarrow \begin{cases} m - p_l.x \leq \delta \\ p_r.x - m \leq \delta \end{cases}$$

Let $S$ be the list of points inside the strip of width $2\delta$ around the separating line, obtained from $Y$ and hence ordered in nondecreasing order of their $y$ coordinate.

✓ **Fact 3:** Let $p$ be a point on this list. For a point $p'$ to have a chance to be closer to $p$ than $\delta$, the point must follow $p$ on list $S$ and the difference between their $y$-coordinates must be less than $\delta$ (why?).



✓ **Fact 4:** The maximum number of points in each $\delta \times 2\delta$ rectangle is 8 (why?).



**Facts 2, 3 & 4** $\Rightarrow$ Each point in the strip needs to be compared with at most 7 points.

```
**ALGORITHM** *EfficientClosestPair*$(X, Y)$
**if** $n \leq 3$
    **return** the minimal distance found by the brute-force algorithm
**else**
    copy the first $\lceil n/2 \rceil$ points of $X$ to $X_L$ and the same points from $Y$ to $Y_L$
    copy the remaining $\lfloor n/2 \rfloor$ points of $X$ to $X_R$ and the same points from $Y$ to $Y_R$
    $\delta_l \leftarrow$ *EfficientClosestPair*$(X_L, Y_L)$
    $\delta_r \leftarrow$ *EfficientClosestPair*$(X_R, Y_R)$
    $\delta \leftarrow \min\{\delta_l, \delta_r\}$
    $m \leftarrow P[\lceil n/2 \rceil - 1].x$
    copy all the points of $Y$ for which $|x - m| < \delta$ into array $S[0 \cdots num - 1]$
    $dminsq \leftarrow \delta^2$
    **for** $i \leftarrow 0$ to $num - 2$ **do**
        $k \leftarrow i + 1$
        **while** $k \leq num$ and $(S[k].y - S[i].y)^2 < \delta^2$
            $dminsq \leftarrow \min\{(S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq\}$
            $k \leftarrow k + 1$
**return** sqrt(dminsq)
```

## Time Complexity Analysis

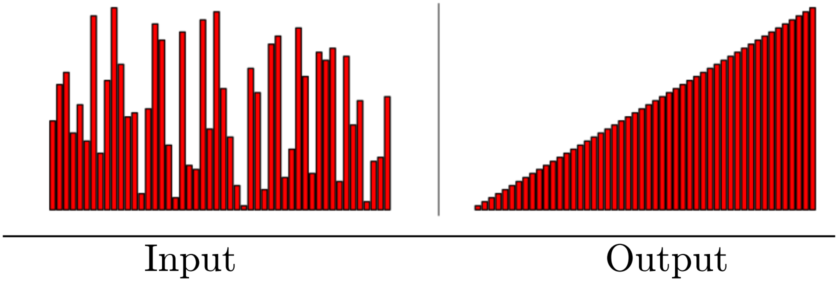
The initial sorting of P by $x$- and $y$-coordinate takes time $O(n \log n)$.
The running time of the remainder of the algorithm satisfies the recurrence

$$T(n) = 2T(n/2) + f(n)$$

where $f(n) \in \Theta(n)$.
$\Rightarrow$ *Master Theorem*: $T(n) \in \Theta(n \log n)$

► It is possible to further improve the running time to $O(n)$ using randomization (interested readers may refer to *Kleinberg*).

## Sorting



**Input description:** a set, $S$, of $n$ items.
**Problem description:** Arrange the items in increasing (or decreasing) order.

**Motivation:**
Sorting is the most fundamental algorithmic problem in computer science.

*Learning the different sorting algorithms is like learning scales for a musician.*

Sorting is the first step in solving a host of other algorithm problems. Indeed, "when in doubt, sort" is one of the first rules of algorithm design.

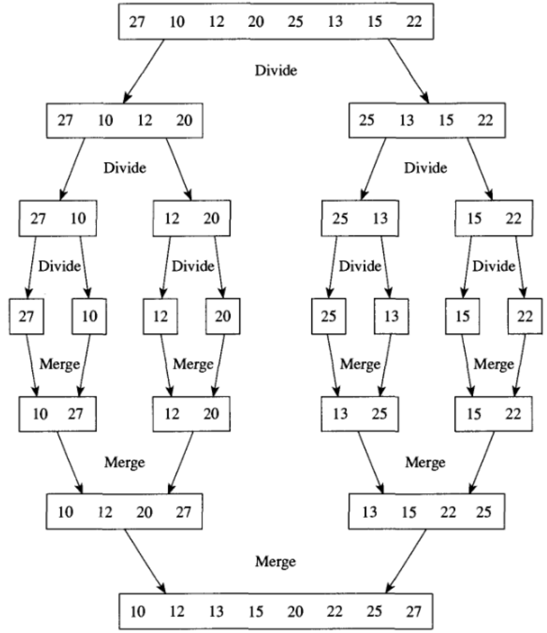
- Many sorting algorithms: bubble-sort, exchange-sort, insertion-sort, shell-sort, heap-sort, merge-sort, quick-sort, ...

We will introduce two efficient methods for sorting:

- Merge sort
- Quick sort

## Merge Sort

1. Divide the array into two subarrays each with $n/2$ items.
2. Conquer (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
3. The combination step ...


27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27

**ALGORITHM 2.2: Mergesort**
**Problem:** Sort $n$ keys in nondecreasing sequence.
**Inputs:** positive integer $n$, array of keys $S$ indexed from 1 to $n$.
**Outputs:** the array $S$ containing the keys in nondecreasing order.

```
void mergesort (int n, keytype S[])
{
const int $h = \lfloor n/2 \rfloor$, $m = n - h$;
keytype $U[1..h]$,$V[1..m]$;
if($n > 1$){
    copy $S[1]$ through $S[h]$ to$U[1]$ through U[h];
    copy $S[h+1]$ through $S[n]$ to $V[1]$ through $V[m]$;
    mergesort($h, U$);
    mergesort($m, V$);
    merge($h, m, U, V, S$);
    }
}
```

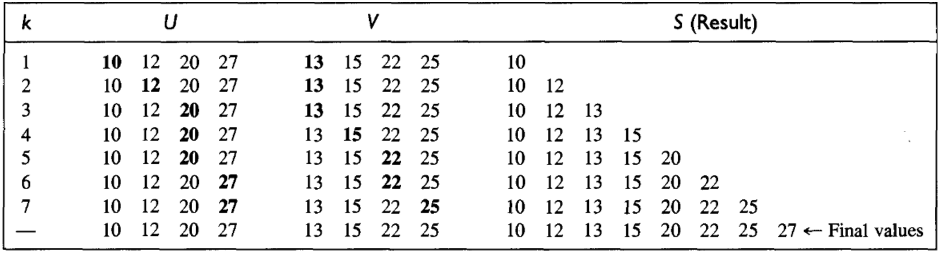
**The combination (merge) step:**
**Problem:** Merge two sorted arrays into one sorted array.
**Inputs:** positive integers $h$ and $m$, array of sorted keys $U$ indexed from 1 to $h$, array of sorted keys $V$ indexed from 1 to $m$.
**Outputs:** an array $S$ indexed from 1 to $h + m$ containing the keys in $U$ and $V$ in a single sorted array.

$\boxed{Stop\&Think}$

$\boxed{\text{There is an algorithm with } O(n) \text{ complexity. What is that?}}$

## ALGORITHM 2.3: Merge

```
**void** *merge* merge (int $h$, int $m$, const keytype $U[\,]$, const keytype $V[\,]$, keytype $S[\,]$)
{
index $i, j, k$;
while ($i <= h$ && $j <= m$) {
    if ($U(i) < V[j]$){
        $S[k] = U[i]$;
        $i++$;
    }
    else{
        $S[k] = V[j]$;
        $j++$;
    }
    $k++$;
}
if ($i > h$)
    copy $V[j]$ through $V[m]$ to $S[k]$ through $S[h+m]$;
else
    copy $U[i]$ through $U[h]$ to $S[k]$ through $S[h+m]$;
}
```



| *k* | *U* | | | | *V* | | | | *S* (Result) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | **10** | 12 | 20 | 27 | **13** | 15 | 22 | 25 | 10 | | | | | | | | |
| 2 | 10 | **12** | 20 | 27 | **13** | 15 | 22 | 25 | 10 | 12 | | | | | | | |
| 3 | 10 | 12 | **20** | 27 | **13** | 15 | 22 | 25 | 10 | 12 | 13 | | | | | | |
| 4 | 10 | 12 | **20** | 27 | 13 | **15** | 22 | 25 | 10 | 12 | 13 | 15 | | | | | |
| 5 | 10 | 12 | **20** | 27 | 13 | 15 | **22** | 25 | 10 | 12 | 13 | 15 | 20 | | | | |
| 6 | 10 | 12 | 20 | **27** | 13 | 15 | **22** | 25 | 10 | 12 | 13 | 15 | 20 | 22 | | | |
| 7 | 10 | 12 | 20 | **27** | 13 | 15 | 22 | **25** | 10 | 12 | 13 | 15 | 20 | 22 | 25 | | |
| — | 10 | 12 | 20 | 27 | 13 | 15 | 22 | 25 | 10 | 12 | 13 | 15 | 20 | 22 | 25 | 27 | ← Final values |

## Worse-case time complexity of Mergesort

**Basic operation:** the comparison that takes place in merge.
**Input size:** $n$, the number of items in the array $S$.

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h+m-1}_{\text{Time to merge, Why?}} \tag{3}$$

Let $n$ be a power of 2, hence

$$W(n) = 2W(n/2) + n - 1, \quad W(1) = 0$$

$\Rightarrow$ *Master Theorem*: $W(n) \in \Theta(n \log n)$

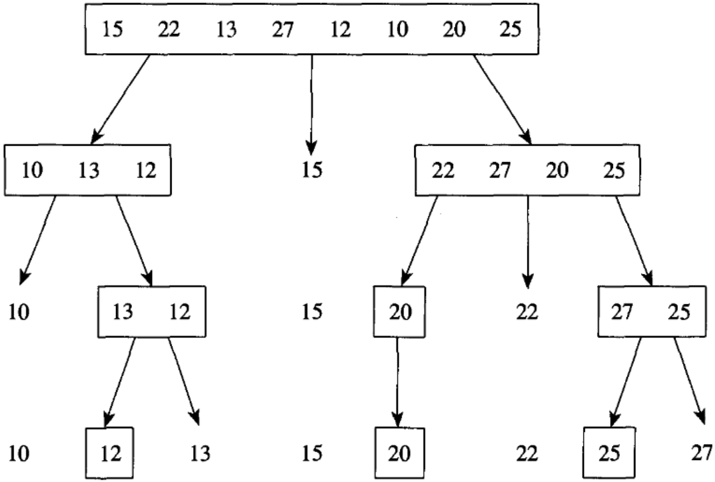
▶ *The reason for the last term in eq. (3)*: Consider algorithm 2.3. The worst case occurs when the loop is exited, because one of the indices–say, $i$–has reached its exit point $h$ whereas the other index $j$ has reached $m-1$, 1 less than its exit point. For example, this can occur when the first $m-1$ items in $V$ are placed first in $S$, followed by all $h$ items in $U$, at which time the loop is exited because $i$ equals $h$.

Example

$U = \{6, 7, 8, 9\}$, $V = \{1, 2, 3, 4, 10\}$ then at exit time: $i = 5$ and $j = 4 \rightarrow$ loop iteration $= i + j - 1 = 8 = m + h - 1$

## Quick Sort

1. Divide: partition the array by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it. The pivot item can be any item, and for convenience we will simply make it the first one.
2. Do the same procedure recursively for the left and right subarrays.
3. No combination needed ...


15 22 13 27 12 10 20 25
10 13 12
15
22 27 20 25
10
13 12
15
20
22
27 25
10
12
13
15
20
22
25
27

**ALGORITHM 2.6: quicksort**
**Problem:** Sort $n$ keys in nondecreasing sequence.
**Inputs:** positive integer $n$, array of keys $S$ indexed from 1 to $n$.
**Outputs:** the array $S$ containing the keys in nondecreasing order.

```
void quicksort (index low, index high)
{
index pivotpoint;
if ($high > low$) {
    partition(low, high, pivotpoint);
    quicksort($low, pivotpoint - 1$);
    quicksort($pivotpoint + 1, high$);
    }
}
```

► If the algorithm were implemented by defining $S$ globally and $n$ was the number of items in $S$, the top-level call to quicksort would be *quicksort*$(1, n)$;

**ALGORITHM 2.7: Partition**

**Problem:** Partition the array $S$ for Quicksort.

**Inputs:** two indices, $low$ and $high$, and the subarray of $S$ indexed from $low$ to $high$.

**Outputs:** $pivotpoint$, the pivot point for the subarray indexed from $low$ to $high$.

```
void partition (index $low$, index $high$, index& $pivotpoint$)
{
index $i$,$j$;
keytype $pivotitem$;
$pivotitem = S[low]$;
$j = low$;
for $(i = low + 1; i <= high; i++)$
      if $(S[i] < pivotitem)${
          $j++$;
          exchange $S[i]$ and $S[j]$;
     }
$pivotpoint = j$;
exchange $S[low]$ and $S[pivotpoint]$;
}
```

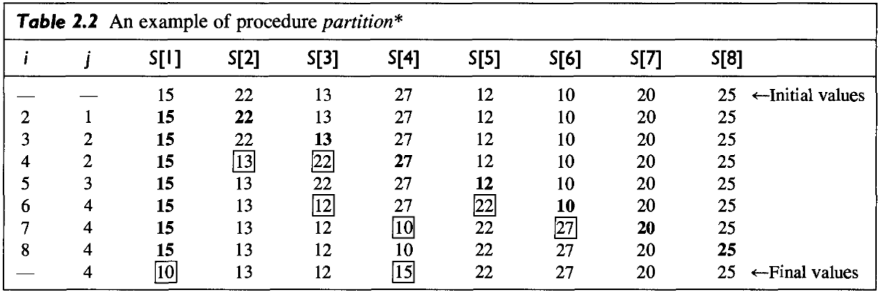


***Table 2.2*** An example of procedure *partition**

| *i* | *j* | S[1] | S[2] | S[3] | S[4] | S[5] | S[6] | S[7] | S[8] | |
|---|---|---|---|---|---|---|---|---|---|---|
| — | — | 15 | 22 | 13 | 27 | 12 | 10 | 20 | 25 | ←Initial values |
| 2 | 1 | **15** | **22** | 13 | 27 | 12 | 10 | 20 | 25 | |
| 3 | 2 | **15** | 22 | **13** | 27 | 12 | 10 | 20 | 25 | |
| 4 | 2 | **15** | $\boxed{13}$ | $\boxed{22}$ | **27** | 12 | 10 | 20 | 25 | |
| 5 | 3 | **15** | 13 | 22 | 27 | **12** | 10 | 20 | 25 | |
| 6 | 4 | **15** | 13 | $\boxed{12}$ | 27 | $\boxed{22}$ | **10** | 20 | 25 | |
| 7 | 4 | **15** | 13 | 12 | $\boxed{10}$ | 22 | $\boxed{27}$ | **20** | 25 | |
| 8 | 4 | **15** | 13 | 12 | 10 | 22 | 27 | 20 | **25** | |
| — | 4 | $\boxed{10}$ | 13 | 12 | $\boxed{15}$ | 22 | 27 | 20 | 25 | ←Final values |

*Items compared are in boldface. Items just exchanged appear in squares.

## Every-Case Time Complexity of Partition

**Basic operation:** the comparison of $S[i]$ with pivotitem.

**Input size:** $n = high - low + 1$, the number of items in the subarray.
Because every item except the first is compared,

$$T(n) = n - 1$$

We are using $n$ here to represent the size of the subarray, not the size of the array $S$. It represents the size of $S$ only when partition is called at the top level.

## Worst-Case Time Complexity of Quicksort

**Basic operation:** the comparison of $S[i]$ with pivotitem in partition.

**Input size:** $n$, the number of items in the array $S$.
Oddly enough, it turns out that the worse case occurs if the array is already sorted in nondecreasing order. For this case, we can wright

$$T(n) = \underbrace{T(0)}_{\text{Time to sort left subarray}} + \underbrace{T(n-1)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

Because $T(0) = 0$, we have the recurrence

$$T(n) = T(n-1) + n - 1, \quad T(0) = 0$$

$$\Rightarrow T(n) = \frac{n(n-1)}{2}$$

By using induction, we can prove (see the proof in the textbook)

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

## Average-Case Time Complexity of Quicksort

**Basic operation:** the comparison of $S[i]$ with pivotitem in partition.

**Input size:** $n$, the number of items in the array $S$.

$$A(n) = \sum_{p=1}^{n} \frac{1}{n} \underbrace{[A(p-1) + A(n-p)]}_{\text{Average time to sort subarrays when pivotpoint is } p} + n - 1$$

We can simply show $\sum_{p=1}^{n} [A(p-1) + A(n-p)] = 2\sum_{p=1}^{n} A(p-1)$, and therefore,

$$A(n) = \frac{2}{n} \sum_{p=1}^{n} A(p-1) + n - 1$$

Multiply by $n$,

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1)$$

## Average-Case Time Complexity of Quicksort

The last equation for $n-1$ gives

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2)$$

Subtract the last two equations

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

which simplifies to

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + 2\frac{n-1}{n(n+1)}$$

If we let $a_n = A(n)/(n+1)$, we have the recurrence

$$a_n = a_{n-1} + 2\frac{n-1}{n(n+1)}, \quad a_0 = 0$$

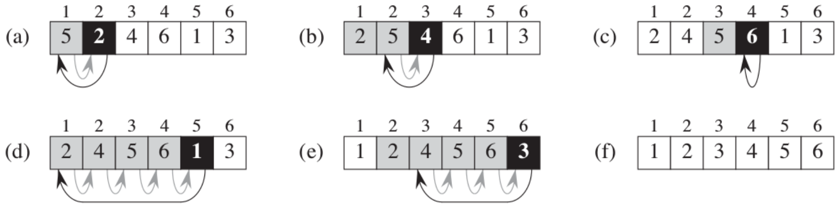
## Average-Case Time Complexity of Quicksort

It can be shown that $a_n \approx 2 \ln n$, and therefore,

$$A(n) \approx (n+1) 2 \ln n \in \Theta(n \log n)$$

## Insertion Sort

- An efficient algorithm for sorting a small number of elements.
- An incremental approach
  - If $A[1...m]$ is sorted how we can sort $A[1...m+1]$?


(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

- Insertion sort works the way many people sort a hand of exam papers.

```
INSERTION-SORT(A)
1  for $j = 2$ to $A.length$
2      $key = A[j]$
3      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.
4      $i = j - 1$
5      while $i > 0$ and $A[i] > key$
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

Prove that:

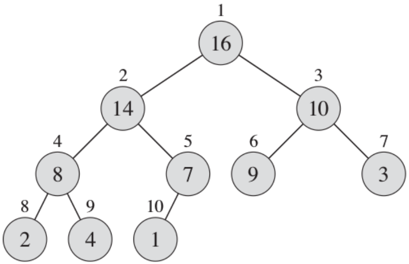
- The best case time complexity is $O(n)$
- The average- and worst-case time complexities are both $\Omega(n^2)$

## Heapsort

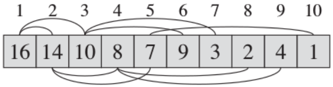
**Heap:** A nearly complete binary tree which satisfies the heap property.

- Nearly complete binary tree: The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
  - ► The tree hight (number of edges from the root to a leaf) = $\lfloor \log n \rfloor$
- Heap property: There are two kinds of binary heaps: max-heaps and min-heaps.
  - max-heap: each parent in the tree $\geq$ it's children $\Rightarrow$ the largest element in a max-heap is stored at the root.
  - min-heap: each parent in the tree $\leq$ it's children $\Rightarrow$ The smallest element in a min-heap is at the root.
- Heap can be implemented by an array.

**Array Representation of Heaps**: For a heap of MaxN nodes, we need an array of size MaxN, plus a variable to store the heap's size.



(a)



(b)

- The root of the tree is $A[1]$, and given the index $i$:

$$PARENT(i) = \lfloor i/2 \rfloor,\ \ LEFT(i) = 2i,\ \ RIGTH(i) = 2i+1$$
$$\Rightarrow A[i] \geq \max\{A[2i], A[2i+1]\} \text{ in max-heap}$$

- Leaves are $A[\lfloor n/2 \rfloor + 1 \cdots n]$. Why?

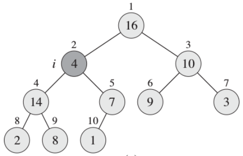
**The Basic Operations on Heaps**:

| | |
|---|---|
| MAX-HEAPIFY$(A, i)$ | $O(\log n)$ |
| BUILD-MAX-HEAP$(A)$ | $O(n)$ |
| MAX-HEAP-INSERT$(A, key)$ | $O(\log n)$ |
| HEAP-EXTRACT-MAX$(A)$ | $O(\log n)$ |
| HEAP-MAXIMUM$(A)$ | $O(1)$ |

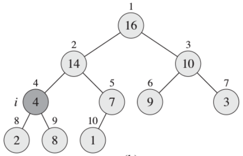
## MAX-HEAPIFY$(A, i)$ $O(\log n)$

- Assumes that the binary trees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but that $A[i]$ might be smaller than its children.
- Verify parental dominance property for $A[i]$. If violated, recursively exchange with the maximum child.

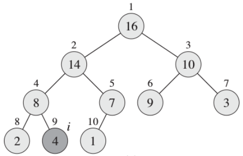
☑$O(\log n)$



(a)



(b)

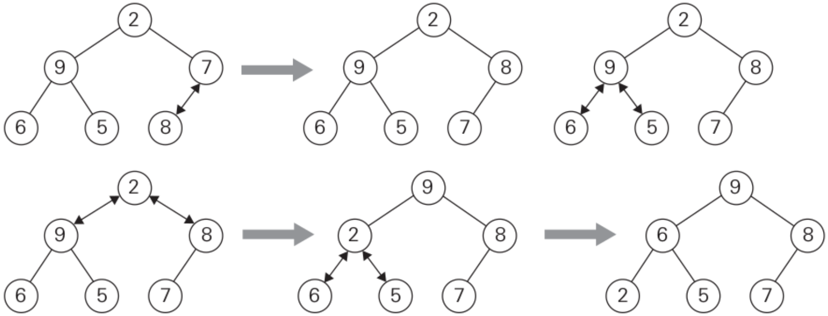


(c)

## BUILD-MAX-HEAP($A$) $O(n)$

- Call MAX-HEAPIFY($A, i$) for $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \cdots, 1$ (all parental nodes)

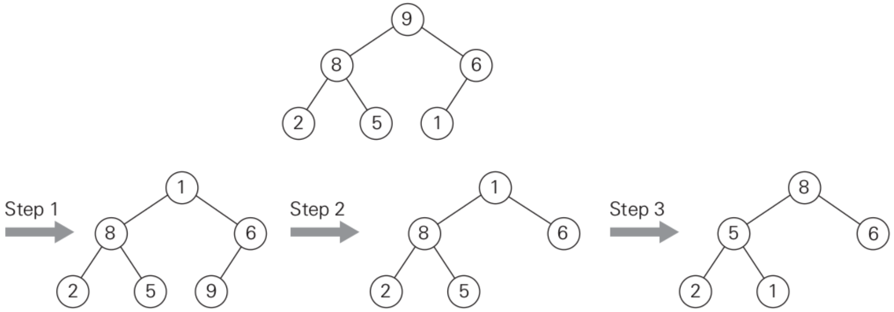
☑$O(n)$, why?



## HEAP-EXTRACT-MAX$(A)$ $O(\log n)$

- Exchange the root's key with the last key $K$ of the heap.
- Decrease the heap's size by 1
- "Heapify" the smaller tree by shifting K down the tree (MAX-HEAPIFY$(A, 1)$)

☑$O(\log n)$



## Heapsort

- **Stage 1** (build the heap):
  Apply BUILD-MAX-HEAP on a given array.
- **Stage 2** (maximum extractions):
  Apply the HEAP-EXTRACT-MAX (root-deletion operation) $n-1$ times to the heap.

$\Rightarrow$ Time complexity of heapsort is $O(n\log n)$.

$$\begin{aligned} T(n) &\lessapprox n + \log n + \log(n-1) + \cdots + \log 2 + \log 1 \\ &= n + \log n! = n + n\log n \\ &\to T(n) \in (n\log n) \end{aligned}$$

## **Example**: Sorting the array 2, 9, 7, 6, 5, 8 by heapsort

| Stage 1 (heap construction) | Stage 2 (maximum deletions) |
|---|---|
| 2 9 **7** 6 5 8 | **9** 6 8 2 5 7 |
| 2 **9** 8 6 5 7 | 7 6 8 2 5 I **9** |
| **2** 9 8 6 5 7 | **8** 6 7 2 5 |
| 9 **2** 8 6 5 7 | 5 6 7 2 I **8** |
| 9 6 8 2 5 7 | **7** 6 5 2 |
| | 2 6 5 I **7** |
| | **6** 2 5 |
| | 5 2 I **6** |
| | **5** 2 |
| | 2 I **5** |
| | 2 |

## Lower bounds for sorting

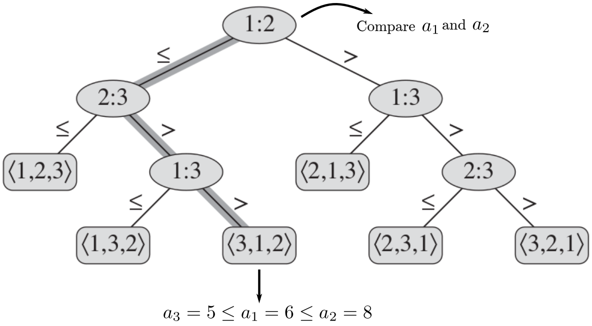
**Comparison Sort Algorithms**

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $< a_1, a_2, \cdots, a_n >$. Here, we assume that all the input elements are distinct $\Rightarrow =$ is useless, and all comparisons have the form $a_i \leq a_j$.

►*The decision-tree model*:
We can view comparison sorts abstractly in terms of **decision trees**. A **decision tree** is a full binary tree (a parent must have 2 children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

**Example**:

- The decision tree for insertion sort operating on three elements.
- The shaded path indicates the decisions made when sorting the input sequence $< a_1 = 6, a_2 = 8, a_3 = 5 >$



**There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.**

## Theorem

*Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.*

## Proof.

It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements. We have

$$n! \leq l \leq 2^h \rightarrow h \geq \log(n!) \in \Omega(n \log n)$$

$\square$

## More on Recursions

## Unbalanced Recursions

## Unbalanced Recursions

Theorem (Akra-Bazzi)

*Consider the recurrence that takes the form*

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n)$$

*where k is a positive integer, $a_i \in R$, $a_i > 0$, and $b_i \in R$, $b_i > 1$ for all $i = 1, \cdots, k$, and $f(n)$ is defined on sufficiently large nonnegative reals and is itself non-negative. Then*

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{f(x)}{x^{p+1}} dx\right)\right)$$

*where $\sum_{i=1}^{k} a_i/(b_i)^p = 1$*

## Recursion Inefficiency

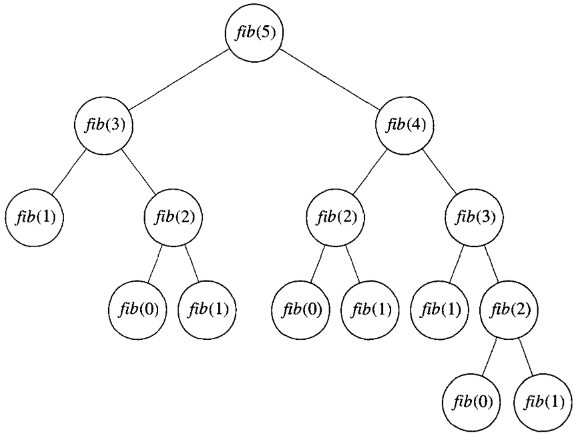
## Fibonacci Sequence

The Fibonacci Sequence $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...)$:

$$f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2} \quad for\ n \geq 2.$$

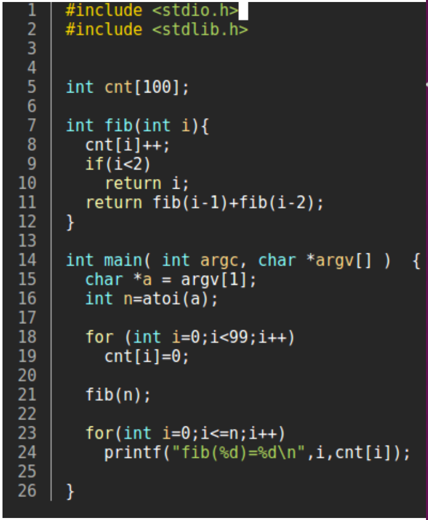
**Algorithm 1.6**
**Problem:** Determine the $n$th term in the Fibonacci sequence.
**Inputs:** a nonnegative integer $n$.
**Outputs:** The $n$th term of the Fibonacci sequence.

```
int fib (int n)
{ if(n <= 1)
    return n;
else
    return fib(n - 1) + fib(n - 2); }
```

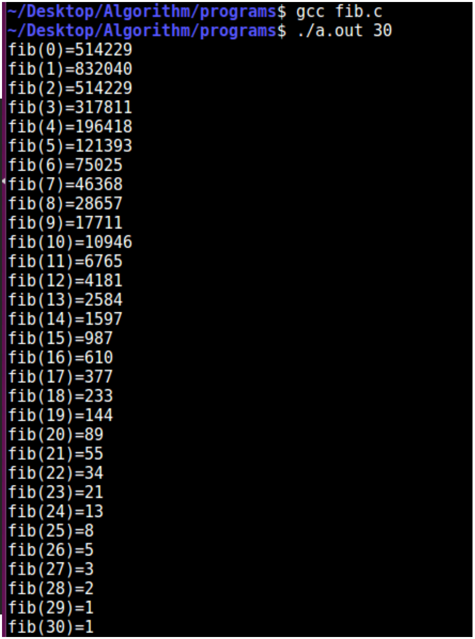


Although the algorithm was easy to create and is understandable, it is extremely inefficient. The divide-and-conquer approach leads to very **efficient** algorithms for some problems, but very **inefficient** algorithms for other problems.

| *n* | *Number of Terms Computed* |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 3 |
| 3 | 5 |
| 4 | 9 |
| 5 | 15 |
| 6 | 25 |



```c
#include <stdio.h>
#include <stdlib.h>


int cnt[100];

int fib(int i){
  cnt[i]++;
  if(i<2)
    return i;
  return fib(i-1)+fib(i-2);
}

int main( int argc, char *argv[] )  {
  char *a = argv[1];
  int n=atoi(a);

  for (int i=0;i<99;i++)
    cnt[i]=0;

  fib(n);

  for(int i=0;i<=n;i++)
    printf("fib(%d)=%d\n",i,cnt[i]);

}
```



```
~/Desktop/Algorithm/programs$ gcc fib.c
~/Desktop/Algorithm/programs$ ./a.out 30
fib(0)=514229
fib(1)=832040
fib(2)=514229
fib(3)=317811
fib(4)=196418
fib(5)=121393
fib(6)=75025
fib(7)=46368
fib(8)=28657
fib(9)=17711
fib(10)=10946
fib(11)=6765
fib(12)=4181
fib(13)=2584
fib(14)=1597
fib(15)=987
fib(16)=610
fib(17)=377
fib(18)=233
fib(19)=144
fib(20)=89
fib(21)=55
fib(22)=34
fib(23)=21
fib(24)=13
fib(25)=8
fib(26)=5
fib(27)=3
fib(28)=2
fib(29)=1
fib(30)=1
```

## Theorem

*If $T(n)$ is the number of terms in the recursion tree corresponding to our algorithm, then, for $n \geq 2$ , $T(n) \geq 2^{n/2}$.*

## Proof.

The proof is by induction on $n$.
Induction base: We need two base cases because the induction step assumes the results of two previous cases. For $n = 2$ and $n = 3$ , we can see that $T(2) = 3 > 2^1 = 2^{2/2}$ and $T(3) = 5 > 2.8323 \approx 2^{3/2}$.
Induction hypothesis: Suppose $T(m) > 2^{m/2}$ for all $2 \leq m < n$.
Induction step:

$$\begin{aligned} T(n) = T(n-1)+T(n-2)+1 &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{n/2-1} = 2^{n/2} \end{aligned}$$

□

## How to Solve the Linear Recurrences?

## Example

Find the solution for the Fibonacci sequence.

$$f_n = f_{n-1} + f_{n-2}, \quad f_n = cr^n$$

$$\to r^n - r^{n-1} - r^{n-2} = 0$$

$$\to r = \frac{1 \pm \sqrt{5}}{2}$$

$$\to f_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n, \quad f_0 = 0, f_1 = 1$$

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n$$

## Example

Find the complexity class of Algorithm 1.6

$$t_n = t_{n-1} + t_{n-2} + 1 \rightarrow t_{n-1} = t_{n-2} + t_{n-3} + 1$$

$$\rightarrow t_n - t_{n-1} = t_{n-1} - t_{n-3} \rightarrow r^n - 2r^{n-1} + r^{n-3} = 0$$

$$\rightarrow r = \frac{1 \pm \sqrt{5}}{2}, 1$$

$$\rightarrow t_n = c_1 \left( \frac{1+\sqrt{5}}{2} \right)^n + c_2 \left( \frac{1-\sqrt{5}}{2} \right)^n + c_3, \quad t_0 = 0, t_1 = 0, t_2 = 1$$

## The Binomial Coefficient

| $\binom{30}{15}$ | 155117520 |
|---|---|
| Input | Output |

**Input description:** Positive integers $n$ and $k$

**Problem description:** find $\binom{n}{k}$. The direct formula is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n$$

But for values of $n$ and $k$ that are not small, direct computation is not feasible because $n!$ is very large even for moderate values of n.

## Theorem

*For all positive integers $k$ and $n$, $0 < k < n$*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

## Proof.

$$\begin{aligned}\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} \\ &= \frac{k(n-1)! + (n-k)(n-1)!}{k!(n-k)!} \\ &= \frac{(n-1)!(k+(n-k))}{k!(n-k)!} = \frac{n!}{k!(n-k)!}\end{aligned}$$

□

We can eliminate the need to compute $n!$ or $k!$ by using this recursive property:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

**Algorithm 3.1**

**Problem:** Compute the binomial coefficient.

**Inputs:** nonnegative integers $n$ and $k$, where $k \leq n$.

**Outputs:** bin, the binomial coefficient $\binom{n}{k}$

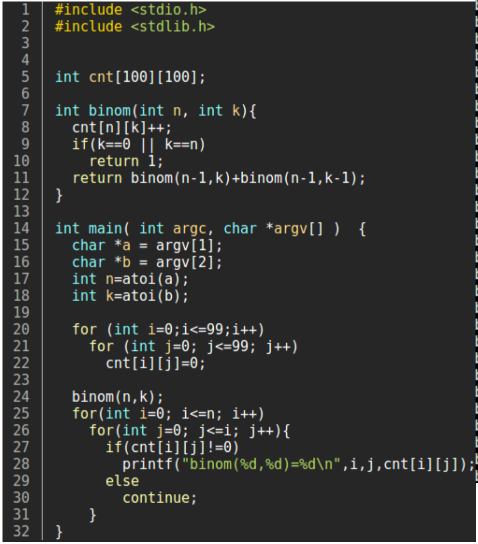
```
int bin (int n, int k)
{ if(k = = 0 || n ==k)
    return 1;
else
    return bin(n - 1, k - 1) + bin(n - 1, k);
}
```

Like Algorithm 1.6 (nth Fibonacci Term, Recursive), this algorithm is **very inefficient**.

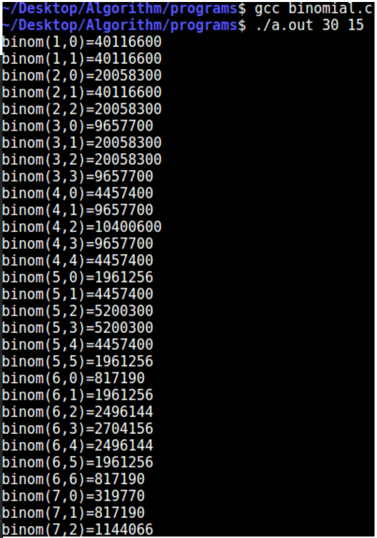
Exercises: Use induction to show that the algorithm computes

$$2\binom{n}{k} - 1$$

terms to determine $\binom{n}{k}$.



```c
#include <stdio.h>
#include <stdlib.h>


int cnt[100][100];

int binom(int n, int k){
  cnt[n][k]++;
  if(k==0 || k==n)
    return 1;
  return binom(n-1,k)+binom(n-1,k-1);
}

int main( int argc, char *argv[] )  {
  char *a = argv[1];
  char *b = argv[2];
  int n=atoi(a);
  int k=atoi(b);

  for (int i=0;i<=99;i++)
    for (int j=0; j<=99; j++)
      cnt[i][j]=0;

  binom(n,k);
  for(int i=0; i<=n; i++)
    for(int j=0; j<=i; j++){
      if(cnt[i][j]!=0)
        printf("binom(%d,%d)=%d\n",i,j,cnt[i][j]);
      else
        continue;
    }
}
```



```
~/Desktop/Algorithm/programs$ gcc binomial.c
~/Desktop/Algorithm/programs$ ./a.out 30 15
binom(1,0)=40116600
binom(1,1)=40116600
binom(2,0)=20058300
binom(2,1)=40116600
binom(2,2)=20058300
binom(3,0)=9657700
binom(3,1)=20058300
binom(3,2)=20058300
binom(3,3)=9657700
binom(4,0)=4457400
binom(4,1)=9657700
binom(4,2)=10400600
binom(4,3)=9657700
binom(4,4)=4457400
binom(5,0)=1961256
binom(5,1)=4457400
binom(5,2)=5200300
binom(5,3)=5200300
binom(5,4)=4457400
binom(5,5)=1961256
binom(6,0)=817190
binom(6,1)=1961256
binom(6,2)=2496144
binom(6,3)=2704156
binom(6,4)=2496144
binom(6,5)=1961256
binom(6,6)=817190
binom(7,0)=319770
binom(7,1)=817190
binom(7,2)=1144066
```

The problem is that the same instances are solved in each recursive call. For example, `bin(n - 1, k - 1)` and `bin(n - 1, k)` both need the result of `bin(n - 2, k - 1)`, and this instance is solved separately in each recursive call.

The divide-and-conquer approach is always inefficient when an instance is divided into two smaller instances that are almost as large as the original instance.