Introduction to Algorithms:

Steps in presenting an algorithmic solution:

1. Describe the algorithm (pseudo code or actual code)
2. If not a trivial solution – "prove" (give an in-depth explanation) to the correctness of your suggested solution
3. Analyze time complexity and if required space complexity
4. Prove time (and space) complexities:
   a. By induction - or -
   b. By contradiction - or -
   c. By construction

Example:

GCD – greatest common divisor

The **greatest common divisor** (gcd, for short) of a and b, written as gcd(a,b), is the largest positive integer that divides both a and b.

The naive algorithm is:

```
Function NaiveGCD($a, b$)
best ← 0
for $d$ from 1 to $a + b$:
  if $d|a$ and $d|b$:
    best ← $d$
return best
```

Remark: operator | means, division with no remainder. For example, *d|a* means that *d* divides *a* with no remainder.

This algorithm's time complexity is O(a+b). We will not discuss this solution further.

A more efficient solution (small improvement) from the naïve algorithm, would be changing the for loop to:

for d from 1 to min(a,b):

This will improve the algorithm's time complexity to O(min(a,b)), which is better than the previous solution. We will not discuss this solution further.

An even more efficient solution would be the Euclidean algorithm. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by the remainder of the division of the larger number by the smaller number. We must prove this assumption.

Step 1: Describe the algorithm (pseudo code or actual code) (a >= b)

```
Function EuclidGCD($a, b$)
if $b = 0$:
  return $a$
$a'$ ← the remainder when $a$ is
   divided by $b$
return EuclidGCD($b, a'$)
```

Step 2 - If not a trivial solution - "prove" the correctness of your suggested solution

Suppose a and b are integers (in which not both are zero), whose gcd has to be determined.
(Since we do not require a full mathematical proof, we will discuss the case where a and b are non-negative, but the theory goes through with essentially no change in case a or b or both, being negative).

The Euclidean algorithms is based on an assumption, in mathematical terms, an assumption is a lemma. We state what the lemma is, and then prove it.
Lemma: if $c=(\mathrm{a} \bmod b)$, then $\gcd(a,b)=\gcd(c,b)$:
Proof: if $c=(\mathrm{a} \bmod b)$, then $b|(a-\mathrm{c})$. So there is a y such that $(a-c)=by$, i.e., $c=(a-by)$.
If d divides both a and b, then it also divides $a-by$ (which is c). Therefore, any common divisor of a and b is also a common divisor of c and b. Similarly, if d divides both c and b, then it also divides a since $\mathrm{a}=c+by$, so any common divisor of c and b is a common divisor of a and b. This shows that the common divisors of a and b are exactly the common divisors of c and b, so, in particular, they have the same greatest common divisor. Thus the greatest common divisor of a and b is the same as the greatest common divisor of b and c. Therefore it is enough if we continue the process with the numbers b and c. Since c is always smaller in absolute value than b, we will reach $\mathrm{c} = 0$ after a finite number of steps.

Step 3: **Analyze time complexity and if required space complexity**.
The time complexity of this solution is $O(\log(\max(\mathrm{a,b})))$. The space complexity of this solution **is** $O(1)$ *

Step 4: **Prove time (and space) complexities**:
The time complexity of this solution is $O(\log(\max(\mathrm{a,b})))$:
In this algorithm, the recursive calls stop, when there is no remainder from the previous division.
Lemma: Each time a division occurs (except maybe the last division), the current number was divided by a number equal or larger than 2.
Proof: if the divider is 1, then there will be no remainder, and the algorithm will stop. Which means, that, except for the last division. All the divisions are done by numbers equal or larger than 2. *Let* $d=max(a,b)$. We have proved that in every iteration, our number was divided at least by 2, which means $O(\log(\max(\mathrm{a,b})))$.
The space complexity of this solution is $O(1)$***.**
*Here we will have to be very specific, because the algorithm itself does not use additional complex data (only a few simple single variables) in computing – thus, $O(1)$ - but the recursive process itself, takes up $O(\log(\max(\mathrm{a.b}))$ space on the runtime stack.

Big – O notation definition:
f(n) = O(g(n)) (f is Big-O of g) if there exist constants N and so that for all n >= N, f(n) <= c·g(n).

Rules in computing asymptotic notation:
1. Multiplicative constants can be omitted. ex. *$7n^3 = O(n^3)$*
2. Smaller terms can be omitted. ex. *$n^2 + n = O(n^2)$*

Common tools used to create algorithms:
- Greedy algorithms
- Divide and conquer
- Dynamic programming

For many algorithmic challenges, there are various solutions, with different time and space complexities. We can divide the types of algorithmic design as following:
- Naïve algorithm
- Algorithm by way of standard tools (greedy, divide and conquer…)
- Optimized algorithm (improvement of previous solution)
- Magic algorithm built on the bases of a unique insight.

(One of the main agendas' of this course, is to find the most efficient solution possible.)

Greedy algorithms:

Greedy is a tool used to build an algorithmic solution. This tool builds a solution, step by step, by choosing, in each step, the option that offers the most obvious and immediate benefit. This policy is built on the assumption (that isn't always true) that if in each step we choose the optimal option, this will lead to an all together best solution.

Each step in the algorithm must create a subproblem of the initial problem. A subproblem is a similar problem of smaller size.

Safe move - definition: a greedy choice is called safe move if there is an optimal solution consisting of this first move. Remark: not all greedy choices are safe moves.

Steps in creating a greedy algorithm solution:

- Make a greedy choice.
- Prove that it is a safe move:
  - Define a lemma – stating that the chosen step is a safe move
  - Prove the lemma
- Reduce into a subproblem
- Solve the subproblem
- Analyze time complexity and if required space complexity
- Prove time (and space) complexities:
  - By induction - or -
  - By contradiction - or -
  - By construction

Input: Given Weights $w_1, \ldots, w_n$ and values $v_1, \ldots., v_n$ of n items; we need to put these items in a knapsack of capacity W.
Input format (example):

n = 3;
{w, v} = {i = 1; w[i] = 10; v[i] = 20}; {i = 2; w[i] = 5; v[i] = 25}; {i = 3; w[i] = 7; v[i] = 28}
W = 19;

We have 3 items, with the given weights and values, and a knapsack of capacity of 19 (weight).
Output: the maximum total value of fractions of the original items that fit into a bag of capacity *W*.
Remark: In Fractional Knapsack - we are allowed to "break" items for maximizing the total value of knapsack. In a brute-force solution, we would try all possible subsets with all possible fractions, but this solution would take too much time.

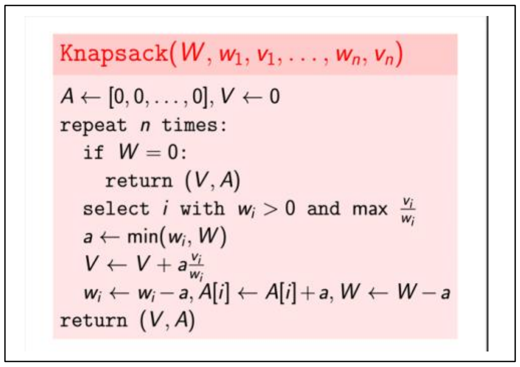
An efficient solution is to use a Greedy approach. Using the greedy approach, we will calculate the ratio ***value/weight*** for each item and sort the items on basis of this ratio. Each new unit is of weight 1 and has a value of v_i/w_i (according to the original *i*). Our greedy choice will be to choose the unit with the highest ratio and add it to the knapsack.

The greedy algorithm would be:

- Compute the ratios
- Sort the ratios in decreasing order
- While knapsack is not full
  - Choose unit i with maximum v_i/w_i
  - If item fits into the knapsack, take all of it.
  - Otherwise take so much as to fill the knapsack and end loop
- Return total value and amounts taken

Pseudo code:

- Compute ratios
- Sort rations into decreasing order, so: you may assume: $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$
- Activate knapsack loop as described ⇒



```
Knapsack($W, w_1, v_1, \ldots, w_n, v_n$)
$A \leftarrow [0, 0, \ldots, 0]$, $V \leftarrow 0$
repeat $n$ times:
  if $W = 0$:
    return $(V, A)$
  select $i$ with $w_i > 0$ and max $\frac{v_i}{w_i}$
  $a \leftarrow \min(w_i, W)$
  $V \leftarrow V + a\frac{v_i}{w_i}$
  $w_i \leftarrow w_i - a$, $A[i] \leftarrow A[i] + a$, $W \leftarrow W - a$
return $(V, A)$
```

In our example:
{w, v} = {i = 1; w[i] = 10; v[i] = 20}; {i = 2; w[i] = 5; v[i] = 25}; {i = 3; w[i] = 7; v[i] = 28}
The ratios are: *{ i = 1: 20/10=2 , i = 2: 25/5=5 , i = 3: 28/7 =4}*
After sorting the ratios: The sorted ratios are: *{ i = 2: 25/5=5, i = 3: 28/7 =4 , i = 1: 20/10 =2}*
W is *19*, so our choices would be:

1. unit2 → value =25/5=5
2. unit2 → value =25/5=5
3. unit2 → value =25/5=5
4. unit2 → value =25/5=5
5. unit2 → value =25/5=5

   remark: no more unit2's
6. unit3 → value =28/7=4
7. unit3 → value =28/7=4
8. unit3 → value =28/7=4
9. unit3 → value =28/7=4
10. unit3 → value =28/7=4
11. unit3 → value =28/7=4
12. unit3 → value =28/7=4

    remark: no more unit3's
13. unit1 → value =20/10=2
14. unit1 → value =20/10=2
15. unit1 → value =20/10=2
16. unit1 → value =20/10=2
17. unit1 → value =20/10=2
18. unit1 → value =20/10=2
19. unit1 → value =20/10=2

sum of values = 67

Proof of safe choice: We will prove that our greedy choice is a safe move by contradiction.
We assume we have an optimal solution, and our knapsack is full – with the highest possible values, and that we have achieved this solution without always choosing our suggested greedy (safe) choice. Which means that in this optimal solution, at some stage, we did not choose a unit with the maximum value to weight ratio (v/w), let's call this unit - unit of item j, or (for short) *unit$_j$*. But we know that when we choose *unit$_j$* there was an item i, that had the highest value to weight (v/w) ratio, *unit$_i$*. We know the value of *unit$_j$* is smaller than the value of *unit$_i$*, so by taking *unit$_j$* out of the knapsack and replacing it with *unit$_i$*, we have increased the knapsacks value, without changing its weight (since we changed a unit of weight 1 with another) of the same weight). This new knapsack will have a larger value, which contradicts that our original choice was the optimal choice. Hence, it is not optimal – and we have proven that our greedy choice is the optimal choice (safe move).

Reduce problem to subproblem: After we choose a greedy choice, we remove it out of the list of units, so we have n-1 items left with n-1 corresponding values.

Solve the subproblem: Keep choosing the maximum value per unit until the capacity *W* is full. You can divide the last unit to a smaller weight unit (smaller than 1) if you need to, to fit the capacity *W*.

Time and space complexity:

Time complexity:
Lemma: the running time of our knapsack solution is *O(nlgn)*
Proof:

- compute ratio is: *O(n)*
- sorting ratio list is: *O(nlgn)*
- while loop - the loop is executed n times = *O(n)*
    - selecting best item on each step is *O(1)*

The time complexity is: compute-ratio-time + sort-time + knapsack-time = *O(n) + O(nlgn) + O(n) = O(nlgn)*

Divide and conquer

In the divide and conquer approach, the problem is divided into smaller, non-overlapping subproblems of the same type. Then each problem is solved independently. We will keep on dividing the subproblems into even smaller subproblems, until eventually we reach a stage in which we cannot divide the subproblem into even smaller subproblems. We than solve all these special, “atomic” subproblems, and join these solutions, until we solve the original whole problem.

Steps in creating a divide and conquer algorithm solution:

- Divide: break problem into non-overlapping subproblems of the same type.
- Conquer: solve subproblems
- Combine results
- Analyze time complexity and if required space complexity
- Prove time (and space) complexities:
    - By induction - or -
    - By contradiction - or -
    - By construction

The first search algorithm introduced in the course is a Linear search in an array. This is not a great example of divide and conquer, though some people regard the recursive version of the algorithm as a simplistic divide and conquer (as was done in this course). We will see better examples later on.

Linear search

Input: an array *A* with *n* elements, and a key *k*.
Output: an index, *i*, where *A[i]=k*. If there is no such *i*, then return *NOT_FOUND*.

Iterative version:

```
LinearSearchIt($A, low, high, key$)

for $i$ from $low$ to $high$:
  if $A[i] = key$:
    return $i$
return NOT_FOUND
```

Recursive version:

```
LinearSearch($A, low, high, key$)

if $high < low$:
  return NOT_FOUND
if $A[low] = key$:
  return $low$
return LinearSearch($A, low + 1, high, key$)
```

Definition: a ***recurrence relation*** is an equation recursively defining a sequence of

For example, Fibonacci can be defined as a recurrence relation:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, \ldots$$

Linear search solution - continued:

Define a corresponding recurrence relation, T

T(n) = T(n-1) + c

T(0) = c

Determine T(n): worst-case runtime

*T(n) = c + c + …. + c = c * n = O(n)*

Which means time complexity = *O(n)*

Space complexity: *O(n)* recursion on runtime call stack

Building a recursive solution:

1. Create a recursive solution
2. Define a corresponding recurrence relation, T
3. Determine T(n): worst-case runtime
4. Optionally, create iterative solution (saves space on runtime stack).

Binary search

Binary search: Searching in a sorted array.

Input: A sorted array *A[low…high] (A[i] < A[i+1])* and a key *k*.

Output: An index, *i,* where *A[i] = k*.

Otherwise, the greatest index *i*, where *A[i] < k.*

Otherwise *(k < A[low]),* the result is *(low — 1).*

Example 1:
Input: [3, 5, 9, 20, 27, 52, 65] and a key 20.
Output: 3 (assuming indexes start from 0)

Example 2:
Input: [3, 5, 9, 20, 27, 52, 65] and a key 7.
Output: 1.

We will solve this problem by using a divide and conquer algorithm

Pseudocode (recursive version):

```
BinarySearch($A$, $low$, $high$, $key$)

if $high < low$:
  return $low - 1$
$mid \leftarrow \left\lfloor low + \frac{high-low}{2} \right\rfloor$
if $key = A[mid]$:
  return $mid$
else if $key < A[mid]$:
  return BinarySearch($A, low, mid - 1, key$)
else:
  return BinarySearch($A, mid + 1, high, key$)
```

Corresponding recurrence relation, T:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

$$T(0) = c$$

The runtime of binary search is **O(logn)**.
Explanation: this can be solved using case 2 of the master theorem (see below).

Space complexity: *O(nlgn)* recursion on runtime call stack

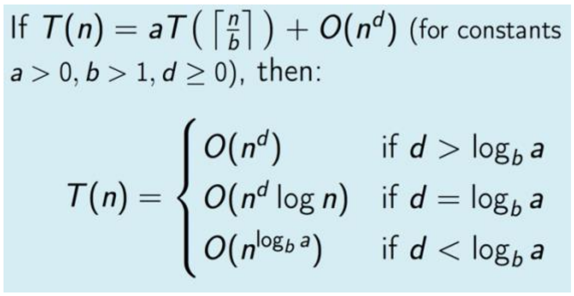
Pseudocode (iterative version):

```
BinarySearchIt($A$, $low$, $high$, $key$)
while $low \leq high$:
  $mid \leftarrow \left\lfloor low + \frac{high-low}{2} \right\rfloor$
  if $key = A[mid]$:
    return $mid$
  else if $key < A[mid]$:
    $high = mid - 1$
  else:
    $low = mid + 1$
return $low - 1$
```

Time complexity is **O(logn)**.

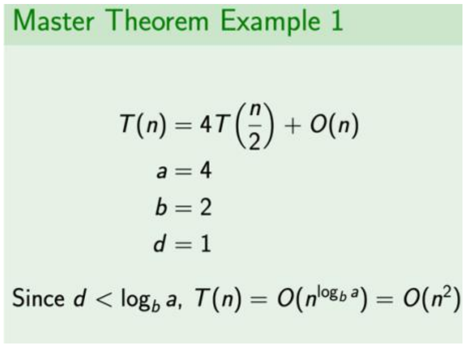
Space complexity: *O(1)* including allocation on runtime call stack

The master theorem: the master method is a formula used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way.



If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ (for constants $a > 0, b > 1, d \geq 0$), then:

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



Master Theorem Example 1

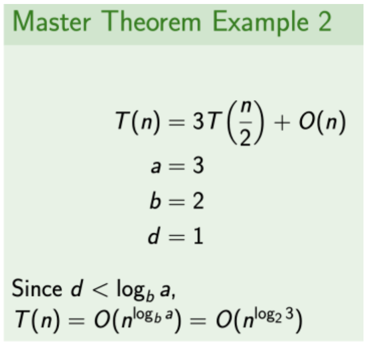
$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

$a = 4$

$b = 2$

$d = 1$

Since $d < \log_b a$, $T(n) = O(n^{\log_b a}) = O(n^2)$



Master Theorem Example 2

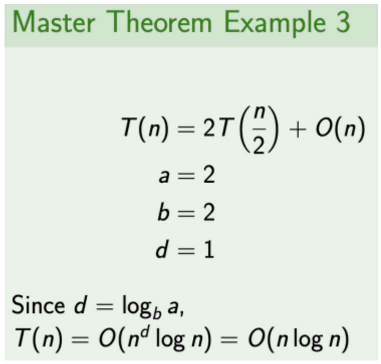
$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$a = 3$

$b = 2$

$d = 1$

Since $d < \log_b a$,
$T(n) = O(n^{\log_b a}) = O(n^{\log_2 3})$



Master Theorem Example 3

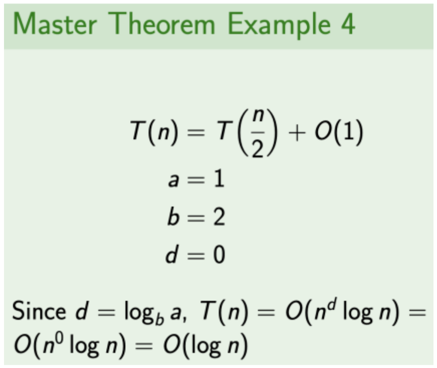
$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$a = 2$

$b = 2$

$d = 1$

Since $d = \log_b a$,
$T(n) = O(n^d \log n) = O(n \log n)$



Master Theorem Example 4

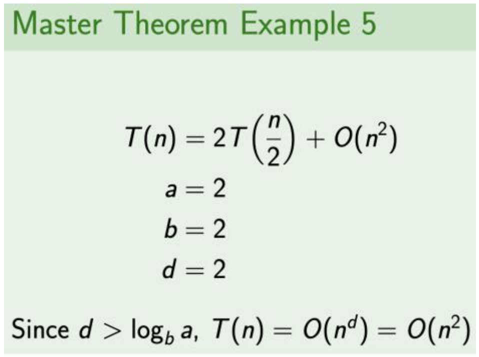
$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$a = 1$

$b = 2$

$d = 0$

Since $d = \log_b a$, $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$



Master Theorem Example 5

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

$a = 2$

$b = 2$

$d = 2$

Since $d > \log_b a$, $T(n) = O(n^d) = O(n^2)$

MergeSort

MergeSort as indicated in its name, breaks-down the array and then merges it together. The concept is very simple:

- Break into 2 sub-arrays, continue breaking until you have arrays of size one.
- Merge 2 sub-list by order(sorting).

Pseudo code:

```
MergeSort($A[1 \ldots n]$)
if $n = 1$:
  return $A$
$m \leftarrow \lfloor n/2 \rfloor$
$B \leftarrow$ MergeSort($A[1 \ldots m]$)
$C \leftarrow$ MergeSort($A[m+1 \ldots n]$)
$A' \leftarrow$ Merge($B, C$)
return $A'$
```

Pseudo continued - the key-point here is the merging strategy. Just loop thought 2 sub-arrays, pick the smaller element, put it into a third-array.

```
Merge($B[1 \ldots p], C[1 \ldots q]$)
{$B$ and $C$ are sorted}
$D \leftarrow$ empty array of size $p + q$
while $B$ and $C$ are both non-empty:
  $b \leftarrow$ the first element of $B$
  $c \leftarrow$ the first element of $C$
  if $b \leq c$:
    move $b$ from $B$ to the end of $D$
  else:
    move $c$ from $C$ to the end of $D$
move the rest of $B$ and $C$ to the end of $D$
return $D$
```

$T(n) = 2T(n/2) + n$
(a=2, b=2, d=1: case 2 master theorem)

Time complexity is **O(nlogn).**
Space complexity is **O(n).**
(since merge sorts most common implementations do not sort in place.

QuickSort

The concept of QuickSort is choosing a pivot, rearranging the array that every elements on the left pivot are smaller than pivot, every elements on the right pivot are bigger than pivot.
The QuickSort algorithm is implemented into 2 steps:

- Choose pivot, re-arrange into A[left] ≤A[pivot] < A[right].
- Keep choosing pivot and re-arrange A[left] and A[right].

The pseudocode – step 1:

```
QuickSort($A, \ell, r$)
if $\ell \geq r$:
    return
$m \leftarrow$ Partition($A, \ell, r$)
{$A[m]$ is in the final position}
QuickSort($A, \ell, m-1$)
QuickSort($A, m+1, r$)
```

The pseudocode – step 2 – rearranging the elements around the pivot – which is the leftmost element of the array:

```
Partition($A, \ell, r$)
$x \leftarrow A[\ell]$    {pivot}
$j \leftarrow \ell$
for $i$ from $\ell+1$ to $r$:
    if $A[i] \leq x$:
        $j \leftarrow j+1$
        swap $A[j]$ and $A[i]$
    {$A[\ell+1 \dots j] \leq x$, $A[j+1 \dots i] > x$}
swap $A[\ell]$ and $A[j]$
return $j$
```

There are many strategies for choosing the pivot:

5. Choose A[0] as a pivot.
6. Choose a random item in the array
7. median of medians algorithm.

The second step of the QuickSort algorithm is how to re-arrange the items in the array around the pivot.
In the following example, we have chosen A[0] as the pivot item (the left most element of the array).

If we prefer choosing a random pivot element:

1. Choose a random pivot *A[k] (l ≤ k ≤ r).*
2. Swap *A[0]* and *A[k]*
3. Execute partition like you did when A[o] was the pivot.

```
RandomizedQuickSort($A, \ell, r$)
if $\ell \geq r$:
    return
$k \leftarrow$ random number between $\ell$ and $r$
swap $A[\ell]$ and $A[k]$
$(m_1, m_2) \leftarrow$ Partition3($A, \ell, r$)
{$A[m_1 \dots m_2]$ is in final position}
RandomizedQuickSort($A, \ell, m_1 - 1$)
RandomizedQuickSort($A, m_2 + 1, r$)
```

Time complexity of the Randomized pivot QuickSort :
Average - **O(nlogn)** in average running time
Worst case - is **O(n^2)**.
Space complexity: (when using in place version) - $O(\log n)$

Comparison of complexities of sorting algorithms:

Algorithm	Best	Average	Worst	Space
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(log(n))

Comparison of complexities of searching algorithms:

Algorithm	Worst	Space
Linear	O(n)	O(n)
Binary	O(n log(n))	O(1) for iterative version O(log(n)) for recursive version

Dynamic programming

- Dynamic programming is the technique of storing repeated computations in memory, rather than recomputing them every time you need them.
- The ultimate goal of this process is to improve runtime.
- Dynamic programming allows you to take less time, at the cost of using more space .

Steps in creating a dynamic programming solution:

- Identify the recurring sub-problems
- Compute a solution for the smallest relevant sub-problem
- Save the results in an array (often called a dp-array – dynamic programming array)
- Build solutions for larger relevant sub-problems, using the previously saved solutions (this avoiding re-computing know solutions)

Continue computing larger and larger solutions until you reach the required solution.

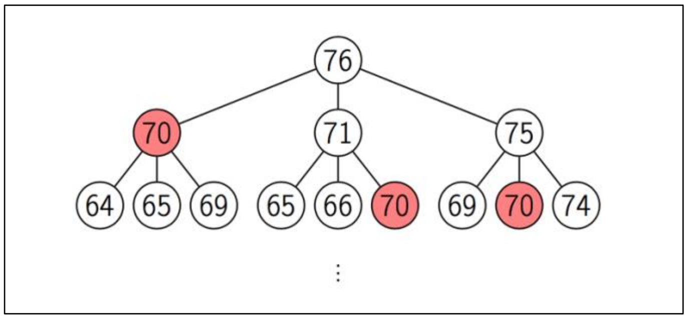
Example 1 - Change **Money** problem:

Let’s say we have 3 types of coin: a 6-cent coin, a 5-cent coin and a 1-cent coin.
We want to give someone 9 cents, with the minimum number of coins possible.

We will solve this problem using Dynamic Programming. The first thing we must do is to find the correct recurrences for this problem.
We can see that the minimum number of coins needed to change 9 cents is the minimum of coins that we need to change 3(= 9–6) or 4(= 9–5) or 8(= 9–1). So we the recurrences we need are:

$$MinNumCoins(9) = \min \begin{cases} MinNumCoins(9-6)+1 \\ MinNumCoins(9-5)+1 \\ MinNumCoins(9-1)+1 \end{cases}$$

The trivial solution to solve this problem would be to use a recursive technique - but it will be very slow. Why slow? Take a look at the recursive tree for changing 76 cents:



This diagram shows only the first three levels of the recursion, in which we already can see, that changing 70 cents will have to be computed at least 3 times, changing 64 cents - at least 2 times, and changing 69 cents - at least 2 times. Obviously if we would have drawn the full recursive tree, we would see many, many more recurring computations of sub-problems. In Dynamic Programming we will memorize sub-problems results for re-use instead of recomputing them – thus saving a lot of time.

For our problem (change money for 9 cents) – we will compute the minimum number of coins needed to change 0, 1, 2, 3,…and save the results into a dp-array. We will continue doing this until we reach 9 cents. The dp-array created will be: dp_array = [0, 1, 2, 3, 4, 1, 1, 2, 3, 4]. So, the answer of how many coins we will need to change 9 cents is the value in dp_array[9] = 4 coins. Following is the solutions pseudo code.

```
DPChange($money, coins$)
$MinNumCoins(0) \leftarrow 0$
for $m$ from 1 to $money$:
  $MinNumCoins(m) \leftarrow \infty$
  for $i$ from 1 to $|coins|$:
    if $m \geq coin_i$:
      $NumCoins \leftarrow MinNumCoins(m - coin_i) + 1$
      if $NumCoins < MinNumCoins(m)$:
        $MinNumCoins(m) \leftarrow NumCoins$
return $MinNumCoins(money)$
```

Example 2 - Discrete Knapsack problem

In the Greedy algorithms part of this summary, we discussed Fractional Knapsack. In the Fractional Knapsack problem, we are allowed to divide the items, and to put "parts" of an item into our knapsack.
In the discrete Knapsack problem, you are only allowed to use whole items, each item is either taken or not.
There are 2 types of Discrete Knapsack: with repetitions and without repetitions.

- With repetitions: there is unlimited number of items of each type, and you can take each item as many times you want.
- Without repetitions: each item appears only once (if we have 2 identical items, they will appear as 2 sperate items), and you can only take these specific items.

Knapsack with repetitions problem:
Input: We have n items with weights: w1, w2, …, wn and values v1, v2, …vn. The knapsack has a total capacity of weight: W.
Output: The maximum value of items whose weight does not exceed W. **Each item can be used any number of times.**
The solution is based on finding the maximum value of n-1 items + the value of an i-th item, or in other words:

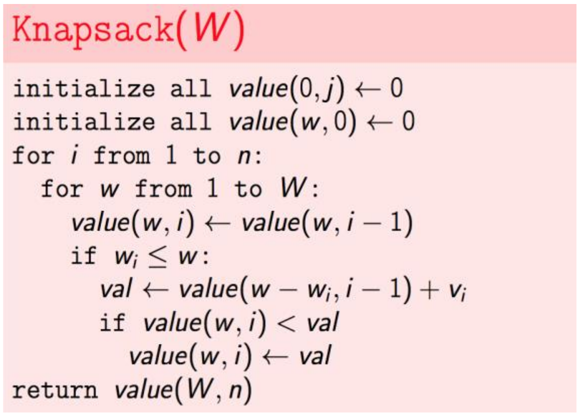
$$value(w) = \max_{i:\, w_i \le w} \{value(w - w_i) + v_i\}$$

The pseudocode for Knapsack with repetitions problem:

```
Knapsack($W$)
$value(0) \leftarrow 0$
for $w$ from 1 to $W$:
    $value(w) \leftarrow 0$
    for $i$ from 1 to $n$:
        if $w_i \le w$:
            $val \leftarrow value(w - w_i) + v_i$
            if $val > value(w)$:
                $value(w) \leftarrow val$
return $value(W)$
```

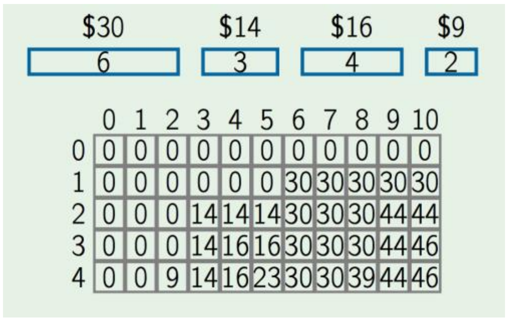
Knapsack without repetitions problem:
Input: We have n items with weights: $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots v_n$. And total capacity weight: W.
Output: The maximum value (W') of items whose weight does not exceed W (W' <= W). **Each item can be used at most once.**
To find out the recurrences of this problem, we need to clarify the main point: Each item is either taken or not. In other words, the maximum value of V_i' is made from maximum value of $V_i' = (v_1 + v_2 + \ldots + v_i)$ and $V_{i-1}' = (v_1 + v_2 + \ldots + v_{i-1})$. V_i' and V_{i-1}' are computed in according to the weights added to the knapsack, ($W_i' = \{w_1, w_2, \ldots, w_i\}$ and $W_{i-1}' = \{w_1, w_2, \ldots, w_{i-1}\}$ accordingly.
Formally: **value(w, i)** = max{ **value(w-wi, i-1) + vi**, **value(w-wi, i-1)** }

So, we need to compute the maximum value of (w = 1, 2,….W, **i = 1**), (w = 1, 2….W, **i = 2**), …. (w= 1, 2, ….W, **i = n**) to get the result **value(w, n)**. We need an 2-D dimensions array to store the cached results.
The pseudocode for Knapsack with repetitions problems.



```
Knapsack($W$)
initialize all $value(0,j) \leftarrow 0$
initialize all $value(w,0) \leftarrow 0$
for $i$ from 1 to $n$:
  for $w$ from 1 to $W$:
    $value(w,i) \leftarrow value(w,i-1)$
    if $w_i \leq w$:
      $val \leftarrow value(w-w_i,i-1)+v_i$
      if $value(w,i) < val$
        $value(w,i) \leftarrow val$
return $value(W,n)$
```

Example:
We have 4 items with weights and values of: (w1, v1)= (6, 30), (w2, v2)= (3, 14), (w3, v3)= (4, 16), (w4, v4)= (2, 9).
By using that strategy, we can build the final table:

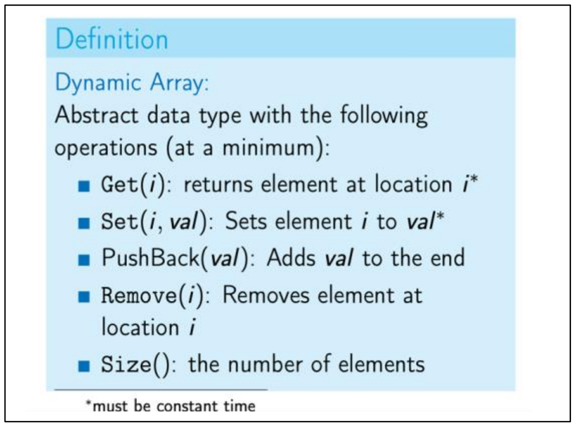

$30
6
$14
3
$16
4
$9
2

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

As we can see, the value[w=10, i=4] is 46. That value is calculated from
max{ value[(w=10) —(wi=2) , i=2] + vi=9, value[w=10, i=2]}
= max{value[8, 2] + 9, value[10, 1] }
= max{ 46, 30} = 46.

Backtracking
After we finish building the table, we can use backtracking to mark which item is used. Like in the above example, value(10, 4) = 46 is chosen from value(8, 2) + 9. So, we can mark item 4 is not used.

Amortized and dynamic array



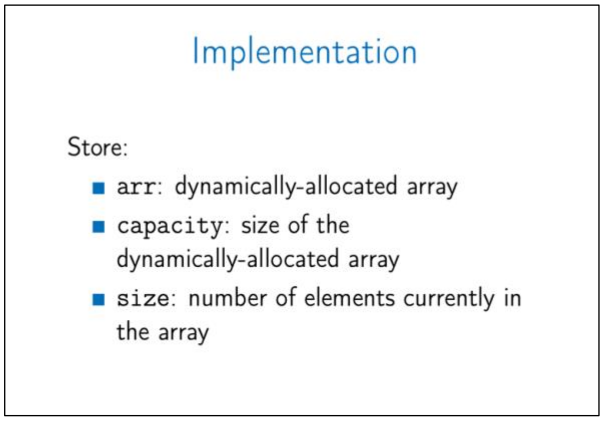
Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- Get(i): returns element at location i*
- Set(i, val): Sets element i to val*
- PushBack(val): Adds val to the end
- Remove(i): Removes element at location i
- Size(): the number of elements

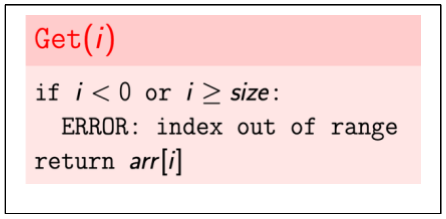
*must be constant time



Implementation

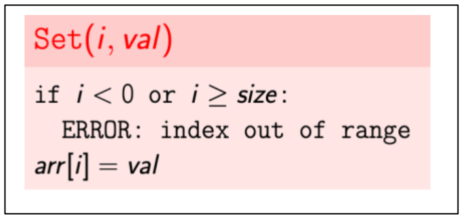
Store:

- arr: dynamically-allocated array
- capacity: size of the dynamically-allocated array
- size: number of elements currently in the array



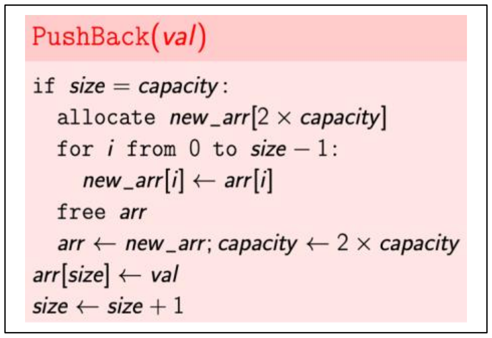
Get(i)

```
if $i < 0$ or $i \geq size$:
  ERROR: index out of range
return $arr[i]$
```



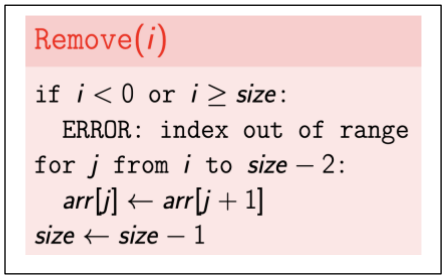
Set(i, val)

```
if $i < 0$ or $i \geq size$:
  ERROR: index out of range
$arr[i] = val$
```



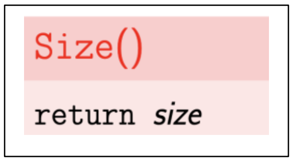
PushBack(val)

```
if $size = capacity$:
  allocate $new\_arr[2 \times capacity]$
  for $i$ from 0 to $size - 1$:
    $new\_arr[i] \leftarrow arr[i]$
  free $arr$
  $arr \leftarrow new\_arr$; $capacity \leftarrow 2 \times capacity$
$arr[size] \leftarrow val$
$size \leftarrow size + 1$
```



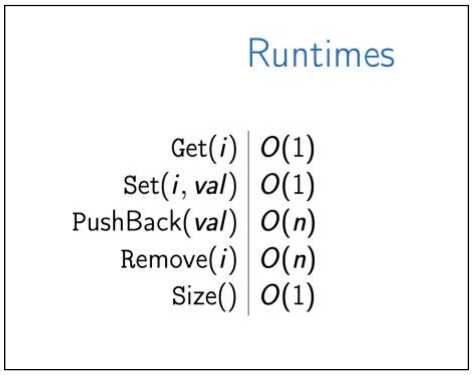
Remove(i)

```
if $i < 0$ or $i \geq size$:
  ERROR: index out of range
for $j$ from $i$ to $size - 2$:
  $arr[j] \leftarrow arr[j + 1]$
$size \leftarrow size - 1$
```



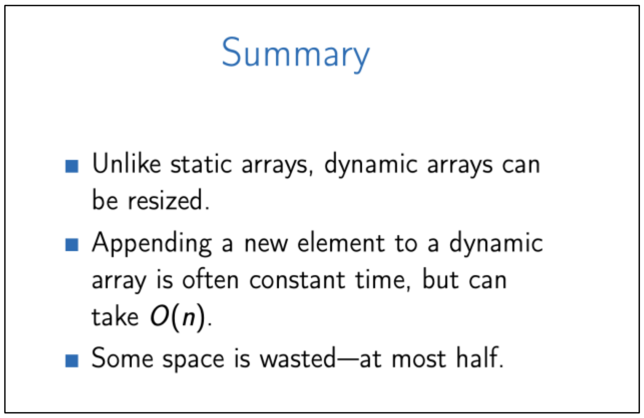
Size()

```
return $size$
```



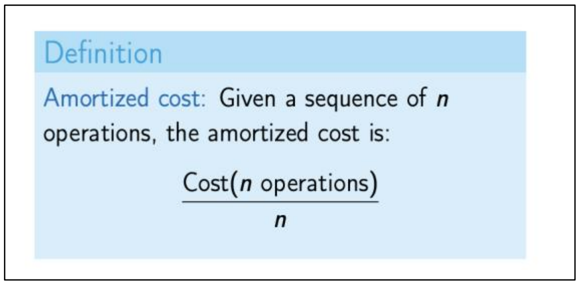
Runtimes

Get(i)	$O(1)$
Set(i, val)	$O(1)$
PushBack(val)	$O(n)$
Remove(i)	$O(n)$
Size()	$O(1)$



Summary

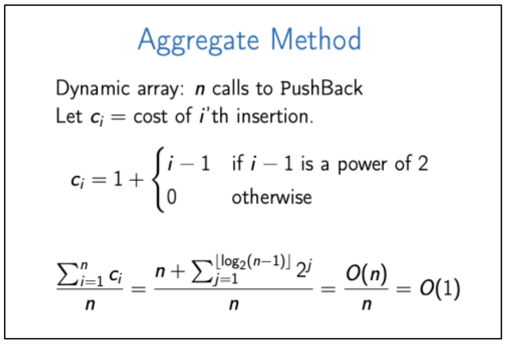
- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take $O(n)$.
- Some space is wasted—at most half.



Definition

Amortized cost: Given a sequence of n operations, the amortized cost is:

$$\frac{\text{Cost}(n \text{ operations})}{n}$$

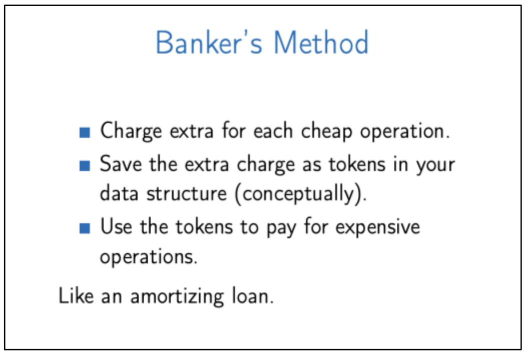


Aggregate Method

Dynamic array: n calls to PushBack
Let c_i = cost of i'th insertion.

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

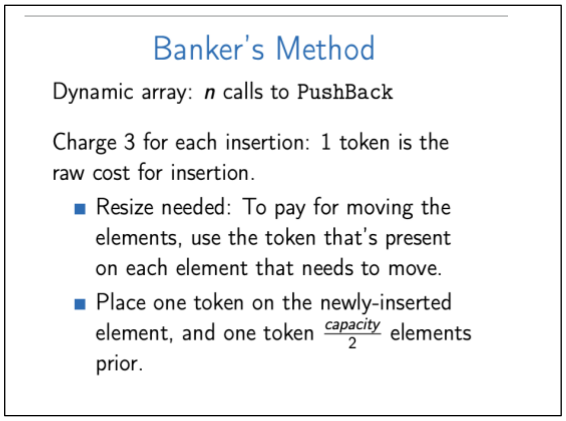
$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n} = O(1)$$



Banker's Method

- Charge extra for each cheap operation.
- Save the extra charge as tokens in your data structure (conceptually).
- Use the tokens to pay for expensive operations.

Like an amortizing loan.

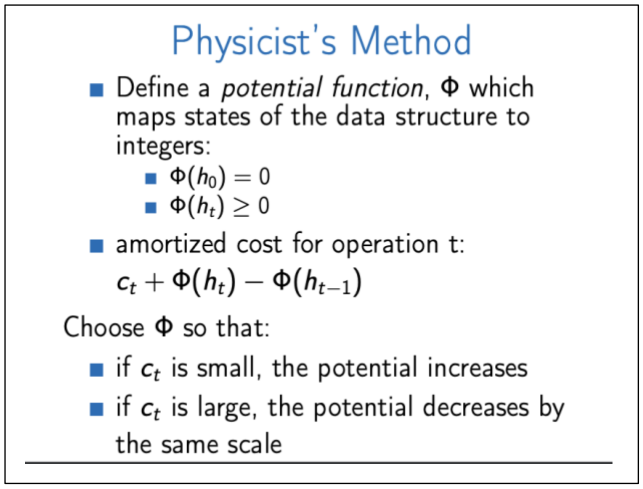


Banker's Method

Dynamic array: n calls to `PushBack`

Charge 3 for each insertion: 1 token is the raw cost for insertion.

- Resize needed: To pay for moving the elements, use the token that's present on each element that needs to move.
- Place one token on the newly-inserted element, and one token $\frac{capacity}{2}$ elements prior.



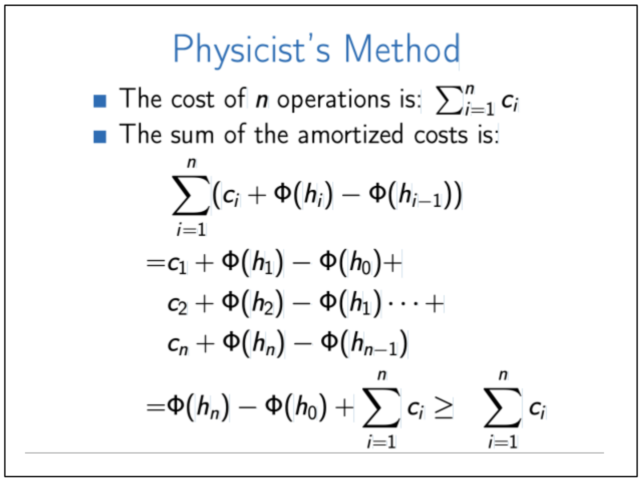
Physicist's Method

- Define a *potential function*, Φ which maps states of the data structure to integers:
  - $\Phi(h_0) = 0$
  - $\Phi(h_t) \geq 0$
- amortized cost for operation t:

  $c_t + \Phi(h_t) - \Phi(h_{t-1})$

Choose Φ so that:

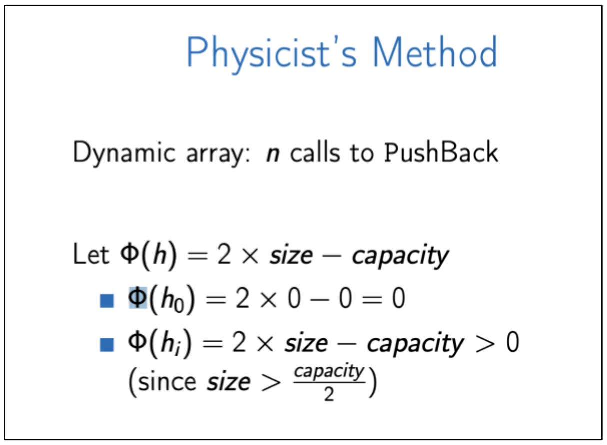
- if c_t is small, the potential increases
- if c_t is large, the potential decreases by the same scale



Physicist's Method

- The cost of n operations is: $\sum_{i=1}^{n} c_i$
- The sum of the amortized costs is:

$$\begin{aligned} &\sum_{i=1}^{n} (c_i + \Phi(h_i) - \Phi(h_{i-1})) \\ =& c_1 + \Phi(h_1) - \Phi(h_0) + \\ & c_2 + \Phi(h_2) - \Phi(h_1) \cdots + \\ & c_n + \Phi(h_n) - \Phi(h_{n-1}) \\ =& \Phi(h_n) - \Phi(h_0) + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i \end{aligned}$$

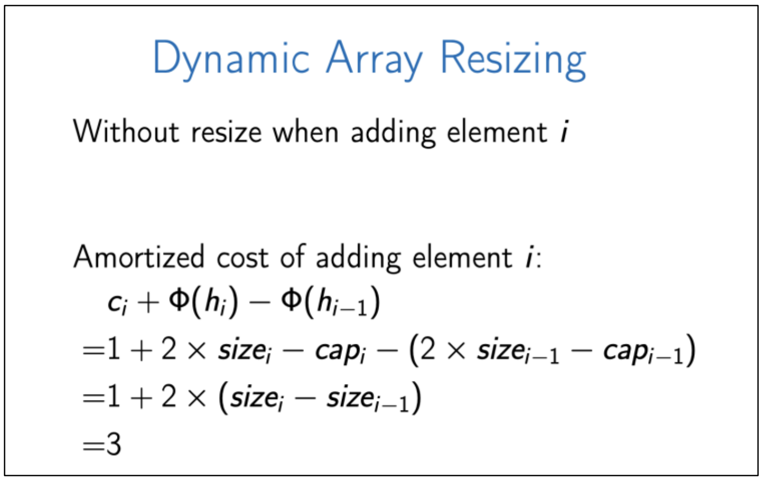


Physicist's Method

Dynamic array: n calls to PushBack

Let $\Phi(h) = 2 \times size - capacity$

- $\Phi(h_0) = 2 \times 0 - 0 = 0$
- $\Phi(h_i) = 2 \times size - capacity > 0$ (since $size > \frac{capacity}{2}$)

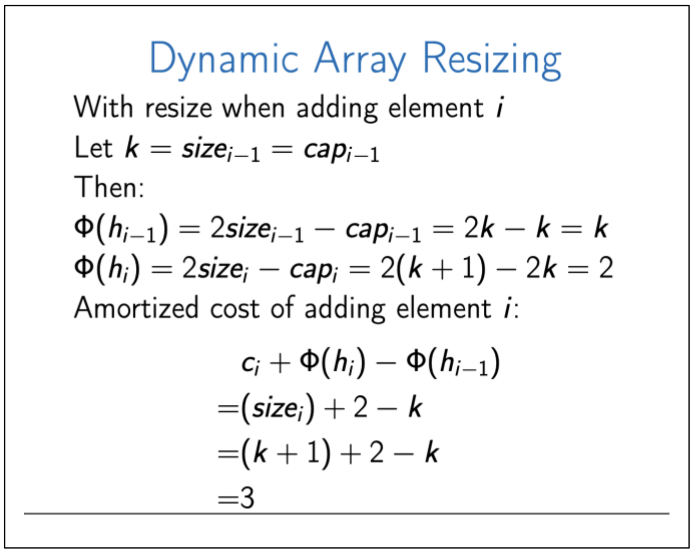


Dynamic Array Resizing

Without resize when adding element i

Amortized cost of adding element i:

$$c_i + \Phi(h_i) - \Phi(h_{i-1})$$
$$=1 + 2 \times size_i - cap_i - (2 \times size_{i-1} - cap_{i-1})$$
$$=1 + 2 \times (size_i - size_{i-1})$$
$$=3$$



Dynamic Array Resizing

With resize when adding element i

Let $k = size_{i-1} = cap_{i-1}$

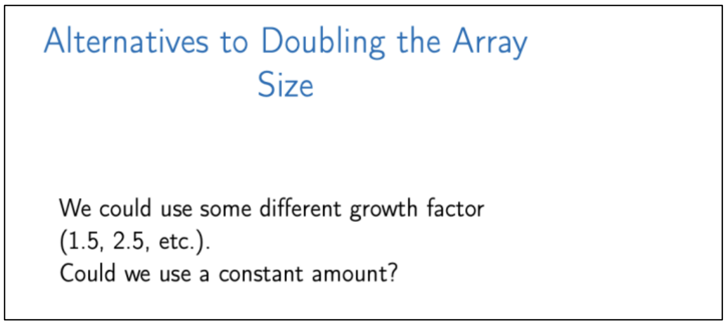
Then:

$\Phi(h_{i-1}) = 2size_{i-1} - cap_{i-1} = 2k - k = k$

$\Phi(h_i) = 2size_i - cap_i = 2(k+1) - 2k = 2$

Amortized cost of adding element i:

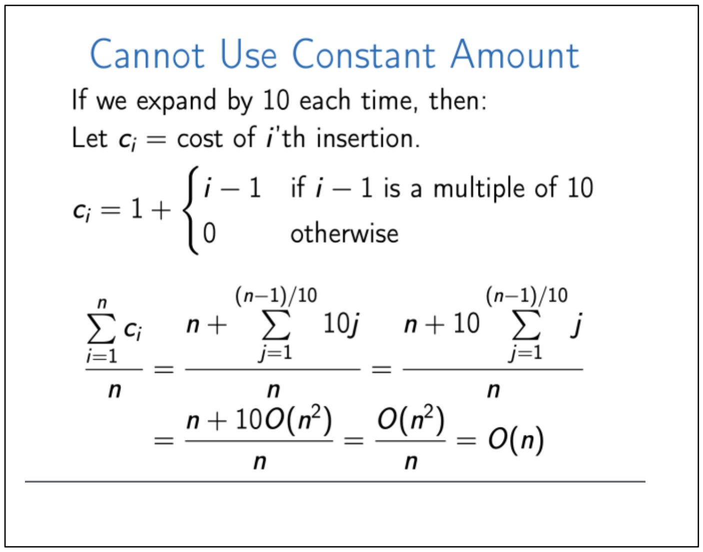
$$c_i + \Phi(h_i) - \Phi(h_{i-1})$$
$$=(size_i) + 2 - k$$
$$=(k+1) + 2 - k$$
$$=3$$



Alternatives to Doubling the Array Size

We could use some different growth factor (1.5, 2.5, etc.).

Could we use a constant amount?



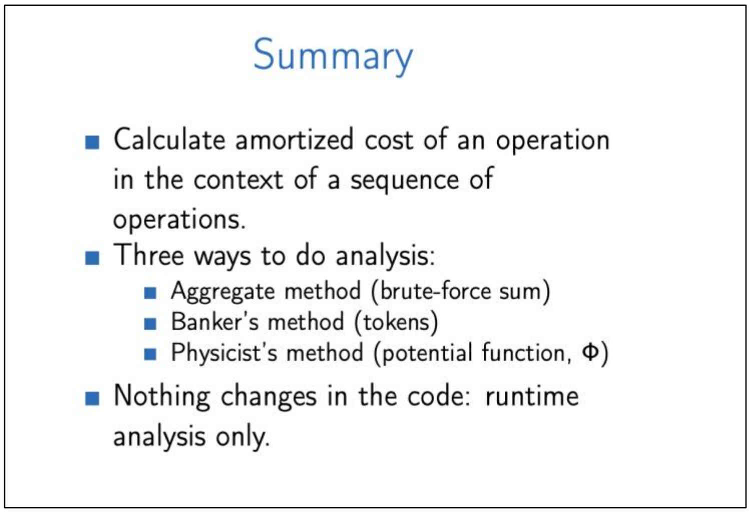
Cannot Use Constant Amount

If we expand by 10 each time, then:

Let $c_i =$ cost of i'th insertion.

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a multiple of } 10 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10 \sum_{j=1}^{(n-1)/10} j}{n}$$
$$= \frac{n + 10O(n^2)}{n} = \frac{O(n^2)}{n} = O(n)$$



Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)
  - Physicist's method (potential function, Φ)
- Nothing changes in the code: runtime analysis only.

Arithmetic Operations

$ab + ac = a(b + c)$ $\qquad a\left(\frac{b}{c}\right) = \frac{ab}{c}$

$\frac{\left(\frac{a}{b}\right)}{c} = \frac{a}{bc}$ $\qquad \frac{a}{\left(\frac{b}{c}\right)} = \frac{ac}{b}$

$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$ $\qquad \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$

$\frac{a - b}{c - d} = \frac{b - a}{d - c}$ $\qquad \frac{a + b}{c} = \frac{a}{c} + \frac{b}{c}$

$\frac{ab + ac}{a} = b + c, \; a \neq 0$ $\qquad \frac{\left(\frac{a}{b}\right)}{\left(\frac{c}{d}\right)} = \frac{ad}{bc}$

Exponent Properties

$a^n a^m = a^{n+m}$ $\qquad \frac{a^n}{a^m} = a^{n-m} = \frac{1}{a^{m-n}}$

$(a^n)^m = a^{nm}$ $\qquad a^0 = 1, \; a \neq 0$

$(ab)^n = a^n b^n$ $\qquad \left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}$

$a^{-n} = \frac{1}{a^n}$ $\qquad \frac{1}{a^{-n}} = a^n$

$\left(\frac{a}{b}\right)^{-n} = \left(\frac{b}{a}\right)^n = \frac{b^n}{a^n}$ $\qquad a^{\frac{n}{m}} = \left(a^{\frac{1}{m}}\right)^n = \left(a^n\right)^{\frac{1}{m}}$

Factoring Formulas

$x^2 - a^2 = (x + a)(x - a)$

$x^2 + 2ax + a^2 = (x + a)^2$

$x^2 - 2ax + a^2 = (x - a)^2$

$x^2 + (a + b)x + ab = (x + a)(x + b)$

Quadratic Formula

Solve $ax^2 + bx + c = 0, \; a \neq 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If $b^2 - 4ac > 0$ - Two real unequal solns.

If $b^2 - 4ac = 0$ - Repeated real solution.

If $b^2 - 4ac < 0$ - Two complex solutions.

Inequalities:

In general, given algebraic expressions *A* and *B*, where *c* is a positive nonzero real number, we have the following properties of inequalities:

Addition property of inequalities:	If $A < B$ then, $A + c < B + c$
Subtraction property of inequalities:	If $A < B$, then $A - c < B - c$
Multiplication property of inequalities:	If $A < B$, then $cA < cB$
	If $A < B$, then $-cA > -cB$
Division property of inequalities:	If $A < B$, then $\frac{A}{c} < \frac{B}{c}$
	If $A < B$, then $\frac{A}{-c} > \frac{B}{-c}$

Also, if A < B and B < C then A < C.

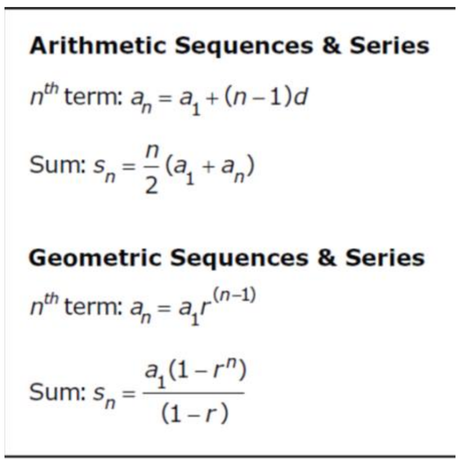
Absolute Value Equations/Inequalitie

If b is a positive number

$|p| = b \quad \Rightarrow \quad p = -b \;\text{ or }\; p = b$

$|p| < b \quad \Rightarrow \quad -b < p < b$

$|p| > b \quad \Rightarrow \quad p < -b \;\text{ or }\; p > b$



Arithmetic Sequences & Series

n^{th} term: $a_n = a_1 + (n-1)d$

Sum: $s_n = \frac{n}{2}(a_1 + a_n)$

Geometric Sequences & Series

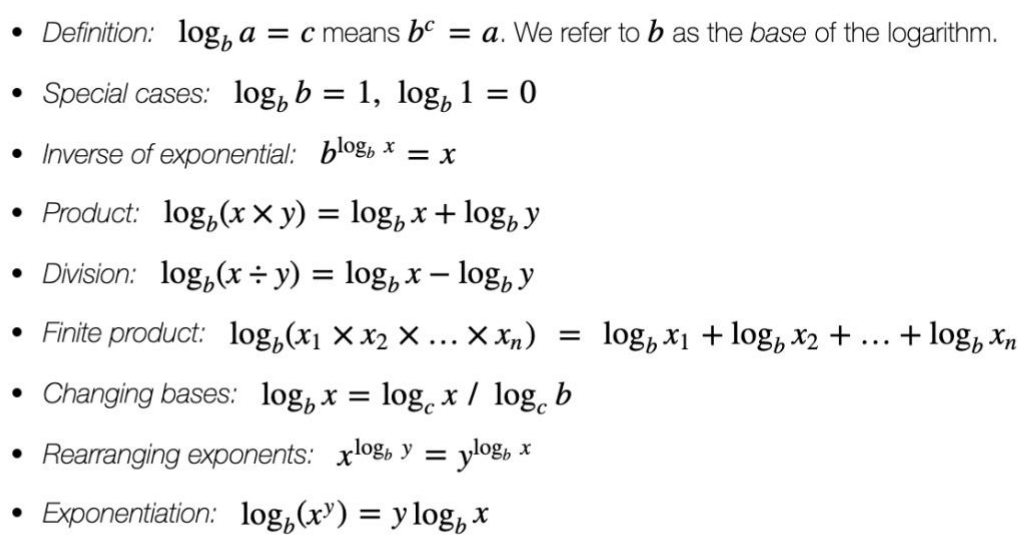
n^{th} term: $a_n = a_1 r^{(n-1)}$

Sum: $s_n = \frac{a_1(1-r^n)}{(1-r)}$

Commonly encountered functions.

function	notation	definition
floor	$\lfloor x \rfloor$	Greatest integer <= x
ceiling	$\lceil x \rceil$	Smallest integer >= x
binary logarithm	lg$x$ or $\log_2 x$	y such that $2^y = x$
natural logarithm	ln$x$ or $\log_e x$	y such that $e^y = x$
common logarithm	$\log_{10} x$	y such that $10^y = x$
harmonic number	H_n	1+1/2+1/3+…+1/n
factorial	$n!$	1×2×3×…×n
binomial coefficeint	$\binom{n}{k}$	$\frac{n!}{k!\,(n-k)!}$

Properties of logarithms.



- *Definition:* $\log_b a = c$ means $b^c = a$. We refer to b as the *base* of the logarithm.
- *Special cases:* $\log_b b = 1,\ \log_b 1 = 0$
- *Inverse of exponential:* $b^{\log_b x} = x$
- *Product:* $\log_b(x \times y) = \log_b x + \log_b y$
- *Division:* $\log_b(x \div y) = \log_b x - \log_b y$
- *Finite product:* $\log_b(x_1 \times x_2 \times \ldots \times x_n) = \log_b x_1 + \log_b x_2 + \ldots + \log_b x_n$
- *Changing bases:* $\log_b x = \log_c x \ / \ \log_c b$
- *Rearranging exponents:* $x^{\log_b y} = y^{\log_b x}$
- *Exponentiation:* $\log_b(x^y) = y \log_b x$

- 1. $c^{\log(a)} = a^{\log(c)}$: take log of both sides.
- $(b^a)^c = b^{ac}$
- $b^a b^c = b^{a+c}$
- $b^a/b^c = b^{a-c}$

Useful formulas and approximations.

Useful formulas and approximations. Here are some useful formulas for approximations that are widely used in the analysis of algorithms.

- *Harmonic sum:* $1 + 1/2 + 1/3 + \ldots + 1/n \sim \ln n$
- *Triangular sum:* $1 + 2 + 3 + \ldots + n = n(n+1)/2 \sim n^2/2$
- *Sum of squares:* $1^2 + 2^2 + 3^2 + \ldots + n^2 \sim n^3/3$
- *Geometric sum:* If $r \neq 1$, then $1 + r + r^2 + r^3 + \ldots + r^n = (r^{n+1} - 1)/(r - 1)$
  - $r = 1/2$: $1 + 1/2 + 1/4 + 1/8 + \ldots + 1/2^n \sim 2$
  - $r = 2$: $1 + 2 + 4 + 8 + \ldots + n/2 + n = 2n - 1 \sim 2n$, when n is a power of 2
- *Stirling's approximation:* $\lg(n!) = \lg 1 + \lg 2 + \lg 3 + \ldots + \lg n \sim n \lg n$
- *Exponential:* $(1 + 1/n)^n \sim e$; $(1 - 1/n)^n \sim 1/e$
- *Binomial coefficients:* $\binom{n}{k} \sim n^k / k!$ when k is a small constant

Asymptotic notations: properties.

- *Reflexivity:* $f(n)$ is $O(f(n))$.
- *Constants:* If $f(n)$ is $O(g(n))$ and $c > 0$, then $c \cdot f(n)$ is $O(g(n))$.
- *Products:* If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) \cdot f_2(n)$ is $O(g_1(n) \cdot g_2(n))$.
- *Sums:* If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max\{g_1(n), g_2(n)\})$.
- *Transitivity:* If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.
- *Symmetry*: $f(n) = O(g(n)) \Leftrightarrow g(n) = O(f(n))$

Divide-and-conquer recurrences. For each of the following recurrences we assume $T(1) = 0$ and that $n/2$ means either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$.

RECURRENCE	$T(n)$	EXAMPLE
$T(n) = T(n/2) + 1$	$\sim \lg n$	*binary search*
$T(n) = 2T(n/2) + n$	$\sim n \lg n$	*mergesort*
$T(n) = T(n-1) + n$	$\sim \frac{1}{2} n^2$	*insertion sort*
$T(n) = 2T(n/2) + 1$	$\sim n$	*tree traversal*
$T(n) = 2T(n-1) + 1$	$\sim 2^n$	*towers of Hanoi*
$T(n) = 3T(n/2) + \Theta(n)$	$\Theta(n^{\log_2 3}) = \Theta(n^{1.58\ldots})$	*Karatsuba multiplication*
$T(n) = 7T(n/2) + \Theta(n^2)$	$\Theta(n^{\log_2 7}) = \Theta(n^{2.81\ldots})$	*Strassen multiplication*
$T(n) = 2T(n/2) + \Theta(n \log n)$	$\Theta(n \log^2 n)$	*closest pair*

Basic asymptotic complexity comparisons:
$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$

Common orders of growth.

NAME	NOTATION	EXAMPLE	CODE FRAGMENT
Constant	O(1)	array access arithmetic operation function call	`op();`

Logarithmic	O(log*n*)	binary search in a sorted array
		insert in a binary heap
		search in a red–black tree
			`for (int i = 1; i <= n; i = 2*i)`
			`  op();`

	$O(\sqrt{n})$		`for (int i = 0; i  * i < n; i++)`
			`  op();`
			- or -
			`for (int i=1, s=1; s<=n; i++, s+=i)`
			`  op();`

Linear	O(*n*)	sequential search	`for (int i = 0; i < n; i++)`
			`  op();`

Linearithmic	O(*n*log*n*)	mergesort
		heapsort
			`for (int i = 1; i <= n; i++)`
			`  for (int j = i; j <= n; j = 2*j)`
			`    op();`

Quadratic	$O(n^2)$	enumerate all pairs
		insertion sort
		grade-school multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`    op();`

Cubic	$O(n^3)$	enumerate all triples
		Floyd–Warshall
		grade-school matrix multiplication
			`for (int i = 0; i < n; i++)`
			`  for (int j = i+1; j < n; j++)`
			`    for (int k = j+1; k < n; k++)`
			`      op();`

Polynomial	is $O(n^k)$ with $k \geq 1$		
Exponential	$O(2^n)$ (Or and constant c $O(c^n)$)	Non greedy Fibonacci numbers	
Factorial	O(n!)	Traveling salesman problem	

Example questions:

Time Complexity Analysis

Question

What is the running time complexity of the following code segment?

```
while n>1 do
  i←n
  while i>1 do
    i←i/2
  n←n/2
```

Solution

הלולאה הפנימית רצה בזמן $\theta(\lg n)$.

הלולאה החיצונית מקטינה את n פי 2 בכל איטרציה. לכן סך כל זמן הריצה הוא:

$$\lg n + \lg\frac{n}{2} + \lg\frac{n}{4} + \lg\frac{n}{8} + \cdots = \sum_{i=0}^{\lg n} \lg\left(\frac{n}{2^i}\right) = \sum_{i=0}^{\lg n} \lg n - \sum_{i=0}^{\lg n} \lg(2^i)$$

$$= (\lg n + 1)\lg n - \sum_{i=0}^{\lg n} i = (\lg n + 1)\lg n - \frac{(\lg n + 1)\lg n}{2}$$

$$= \frac{(\lg n + 1)\lg n}{2} = \theta((\lg n)^2)$$

Question

What is the running time complexity of the following code segment?

```
for i←⌊lg(n)⌋ to ⌊n^0.5⌋ do
  for j←1 to i do
    k←2
    while k<n^2 do
      k←k*k
```

Solution

נתחיל מהלולאה הפנימית. K מוכפל בעצמו בכל איטרציה. כלומר: ערכו באיטרציה ה-i הוא: $2^{(2^i)}$. תוך כמה איטרציות יגיע ערכו לערך n^2?

$$2^{(2^i)} = n^2$$

נוציא לוגריתם משני האגפים:

$$2^i = 2\lg n$$

נחלק את שני האגפים ב-2 ונוציא, שוב, לוגריתם משני האגפים:

$$i - 1 = \lg\lg n$$

זמן הריצה האסימפטוטי של הלולאה הפנימית הוא לוגריתמי.

הלולאה הבאה (לולאת for ש-j הוא האינדקס שלה) מתבצעת בדיוק i פעמים. מכיוון שגוף הלולאה מתבצע בזמן $O(\lg n)$, הרי שהלולאה כולה רצה בזמן $O(i\lg\lg n)$.

הלולאה החיצונית מהווה אתגר קצת יותר גדול, ומצריכה שימוש בנוסחה לסכום של סדרה חשבונית:

$$\sum_{i=\lg n}^{\sqrt{n}} i\lg\lg n = \lg\lg n \sum_{i=\lg n}^{\sqrt{n}} i = \lg\lg n\left(\sum_{i=1}^{\sqrt{n}} i - \sum_{i=1}^{\lg n - 1} i\right)$$

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

$$= \theta(n\lg\lg n)$$

Question

What is the running time complexity of the following code segment?

```
F(n)
sum←0
if n=1
   then return 1
for i←0 to ⌊n/2⌋ do
   sum←sum+i
sum←sum+F(⌊n/2⌋)
for i←n downto n-⌊n/7⌋ do
   sum←sum+i
sum←sum+F(⌊n/7⌋)
return sum
```

פתרון

נוסחת הנסיגה היא:

$T(n)=T(n/2)+T(n/7)+n$

כדי לפתור אותה נדרש מעט תחכום.

$T(n/2)+n\leq T(n/2)+T(n/7)+n$

מצד שני, מכיוון ש T(n) הוא לפחות n, הפתרון הוא לפחות לינארי ולכן:

$T(n/2)+T(n/7)\leq T(n/2+n/7)\leq T(3n/4)$

אם כך, אנחנו יודעים ש:

$T(n/2)+n\leq T(n/2)+T(n/7)+n\leq T(3n/4)+n$

נפתור את שתי נוסחאות הנסיגה:

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

$T(n)= T(3n/4)+n$

לשתיהן אותו פתרון:

T(n)=Θ(n)

ולכן זהו גם הפתרון של נוסחת הנסיגה ממנה התחלנו, וזמן הריצה של האלגוריתם הוא: Θ(n)

Greedy Algorithm

Question

Prove that the greedy algorithm for solving the money change problem always returns the minimum number of coins when the coins are: 1,2,5,10

הוכיחו שהאלגוריתם החמדן לפתרון בעיית החזרת העודף מחזיר תמיד מספר מינימלי של מטבעות כשהמטבעות הן: 1,2,5,10.

פתרון

תובנה ראשונה: כל אלגוריתם שפותר את הבעיה ישתמש במטבע יחיד של 1, לכל היותר, כי שני מטבעות של 1 ניתן להחליף במטבע של 2.תובנה שנייה: כל אלגוריתם שפותר את הבעיה ישתמש בשני מטבעות של 2, לכל היותר, כי שלושה
מטבעות של 2 אפשר להחליף במטבע של 5 ומטבע של 1.תובנה שלישית: כל אלגוריתם שפותר את הבעיה ישתמש במטבע יחיד של 5, לכל היותר, כי שני מטבעות של 5 ניתן להחליף במטבע יחיד של 10.
כדי להוכיח שהאלגוריתם החמדן מחזיר מינימום מטבעות, נראה שכל אלגוריתם שבוחר אחרת מהחמדן, לא מחזיר מספר מינימלי של מטבעות. כדי לעשות זאת נמיין את המטבעות שהאלגוריתם האחר מחזיר, מהגדול לקטן, ונשווה את הרשימה לרשימת המטבעות שמחזיר האלגוריתם החמדן.נעבור על שתי הרשימות מטבע מטבע (מהגדול לקטן) עד שנגיע למקום ברשימה בו יש הבדל בין הבחירות של שני האלגוריתמים. נניח שהאלגוריתם החמדן מחזיר מטבע X והאלגוריתם האחר מחזיר מטבע Y. ברור ש- $X>Y$ כי האלגוריתם החמדן מחזיר את המטבע הגדול ביותר האפשרי. נניח ש X=10. במקרה זה Y הוא לכל היותר 5. והאלגוריתם האחר יכול להחזיר (לאור התובנות מראשית ההוכחה): 5,2,2,1, לכל היותר. מכיוון ששני האלגוריתמים מחזירים אותו סכום, הם לא יכולים להחזיר מטבעות נוספים. האלגוריתם האחר מחזיר שלושה מטבעות יותר מהחמדני.

נניח ש x=5. במקרה זה y הוא לכל היותר 2. והאלגוריתם האחר יכול להחזיר (לאור התובנות מראשית ההוכחה) 2,2,1 לכול היותר. מכיוון ששני האלגוריתמים מחזירים אותו סכום, הם לא יכולים להחזיר מטבעות נוספים. האלגוריתם האחר מחזיר שני מטבעות יותר מהחמדני.

נניח ש X=2. במקרה זה Y הוא לכל היותר 1. והאלגוריתם האחר לא יכול להחזיר 2, והסכום שהוא מחזיר חייב להיות קטן מהסכום שמחזיר האלגוריתם החמדן. זו סתירה להנחה ששני האלגוריתמים מחזירים אותו סכום.לכן, כל אלגוריתם שמחזיר מטבעות שונים מהאלגוריתם החמדן, לא יחזיר מינימום של מטבעות, ומכאן שהאלגוריתם החמדן מחזיר מינימום

של מטבעות.

Question

In the computer science department, lessons are held on Sundays in *n* different courses. Each course I ($1 \le i \le n$) has a start time, s_i (start) and an end time f_i (finish).

The department secretariat wants to include as much courses as possible in the auditorium, because this is the most comfortable class (comfortable chairs, modern multimedia equipment, electronic board, etc.).

It is clear that two courses that take place at the same time or that their hours intersect (a lesson's finish time is before other lesson's start time) cannot take place in the same classroom.

Describe a greedy algorithm to solve the problem

בחוג למדעי המחשב מתקיימים בימי א שיעורים ב- *n* קורסים שונים. לכל קורס i ($1 \le i \le n$) יש זמן החלה s_i (start) וזמן סיום f_i (finish).

במזכירות החוג רוצים לשבץ את מירב הקורסים באודיטוריום, כי זוהי הכיתה הנוחה ביותר (כיסאות נוחים, ציוד מולטימדיה חדיש, לוח אלקטרוני וכד'). ברור כי שני קורסים המתקיימים באותן שעות, או ששעותיהם חותכות זו את זו (כלומר זמן ההתחלה של קורס אחד מוקדם מזמן הסיום של קורס אחר), לא יכולים להתקיים באותו אולם.

תארו אלגוריתם חמדן הפותר את הבעיה.

פתרון

האלגוריתם החמדן יבחר בכל שלב את הקורס שנגמר הכי מוקדם ושלא מתנגש עם הקורסים שכבר נבחרו .כדי להוכיח שהבחירה החמדנית נכונה, נשווה את רשימת הקורסים שבחר האלגוריתם החמדן לרשימת הקורסים שבחר אלגוריתם אחר (ממוינת לפי שעת סיום הקורס), ונראה שניתן לשנות את בחירת האלגוריתם האחר לבחירה החמדנית בלי להקטין את מספר הקורס המשובצים לאודיטוריום. נסתכל על המקום הראשון ברשימה בו יש הבדל בין שני האלגוריתמים. הקורס שבחר האלגוריתם האחר מסתיים לא לפני האלגוריתם שבחר האלגוריתם החמדן. לכן אם נחליף את בחירת האלגוריתם האחר לבחירת החמדן הרשימה תישאר רשימה של קורסים שלא מתנגשים זה עם זה: הקורס "החמדן" בוודאי לא מתנגש עם הקורסים שלפניו בגלל שהוא נגמר לא אחרי הקורס "האחר" הרי שהוא גם לא יתנגש עם הקורסים שאחריו (כי "האחר" לא התנגש איתם). כלומר, כל אלגוריתם אחר יכול לשנות את בחירתו לבחירה החמדנית. האלגוריתם החמדן משבץ מקסימום של קורסים.

Question

In the computer science department, lessons are held on Wednesdays in *n* different courses. We'll mark the collection of courses in S={1,2,…,n}. each course has a start time s_i and a finish time f_i.

The department secretariat wants to include the courses to the minimum number of classrooms.   It is clear that two courses that take place at the same time or that their hours intersect (a lesson's finish time is before other lesson's start time) cannot take place in the same classroom.

In order to the solve the problem, the following algorithm was suggested:

בחוג למדעי המחשב מתקיימים בימי רביעי שיעורים ב – n קורסים שונים.

נסמן את קבוצת הקורסים ב – {1,2,…,n} = S.   לכל קורס i יש זמן התחלה s_i וזמן סיום f_i.

במזכירות החוג רוצים לשבץ את הקורסים למספר מינימלי של אולמות.

ברור כי שני קורסים המתקיימים באותן שעות, או ששעותיהם חופפות חלקית, לא יכולים להתקיים באותו אולם.

כדי לפתור את הבעיה הוצע האלגוריתם הבא:

```
MINIMUM-NUMBER-OF-HALLS (s, f)
1.  count ← 0
2.  while S ≠ ϕ
3.  do count ← count + 1
4.      A ←GREEDY-ACTIVITY-SELECTOR (s, f)
5.      place the courses of A in hall  No. count
6.      S ← S – A
```

The GREEDY-ACTIVITY-SELECTOR (*s*, *f*) is an algorithm that places maximum courses to a single classroom.

Prove or refute: the suggested algorithm will always succeed to set a given collection of courses to a minimum number of classrooms.

האלגוריתם GREEDY-ACTIVITY-SELECTOR (*s*, *f*) הוא האלגוריתם המשבץ מקסימום קורסים לאולם יחיד.

הוכיחו או הפריכו: האלגוריתם תמיד יצליח לשבץ קבוצה נתונה של קורסים למספר מינימלי של אולמות.

פתרון

האלגוריתם ייכשל, למשל, עבור הקלט הבא:

f1=1100 s1=1500 f2=0900 s2=1200 f3=0800 s3=1000 f4=1300 s4=1400

הוא ישבץ את קורס 3 ו-4 בכיתה, אחר כך ישבץ את קורס 2 בכיתה, ולסיום ישבץ את קורס 1 בכיתה.

סה"כ 3 כיתות. ניתן לשבץ את הקורסים בשתי כיתות: קורס 1 וקורס 3 בכיתה אחת. קורס 2 וקורס 3 בכיתה שניה.

האלגוריתם ימיין את הקטעים לפי נקודת ההתחלה שלהם.

אם נקודת ההתחלה הקטנה ביותר גדולה מ-0 אז אין כסוי.

B יאותחל לנקודת הסיום של הקטע הראשון. האלגוריתם יעבור על כל הקטעים, לפי הסדר, ובכל איטרציה יעדכן את *B* כך שכל הקטעים שנסרקו עד עתה מכסים, ללא "חורים", את הקטע [0,*B*].

אם בסיום האלגוריתם *B*<1 אז אין כסוי, אחרת יש כסוי.

זמן הריצה הוא זמן הריצה של אלגוריתם המיון.

```
cover(A)
//A is the segment array
use merge sort to sort A according to s.
if A[1].s>0
then return false
B←A[1].f
for i←2 to length[A] do
  if A[i].s>B
  then return false
  B←max(B,A[i].f)
return B≥1
```

Divide and Conquer

Question

Reverse in an array A of size n is a pair of indexes i and j (0<i<j≤n), such that A[j]<A[i].

In other words, **reverse** is a a pair of elements that the greater one appears before the lower one in the array order.

Write a **divide and conquer** algorithm that calculates the number of reverses in a given array. Assume all array elements are different.

היפוך במערך *A* בגודל *n* הוא זוג אינדקסים *i* ו-*j* $(0<i<j\leq n)$ כך ש: $A[j]<A[i]$.

במילים פשוטות: היפוך הוא זוג איברים שהגדול שבהם מופיע לפני הקטן.

כתבו אלגוריתם, שפועל בשיטת הפרד ומשול, המחשב את מספר ההיפוכים במערך נתון. הניחו שכל האיברים במערך שונים זה מזה.

רמז: מיון מיזוג.

פתרון

נספור לכל איבר כמה איברים הקטנים ממנו נמצאים אחריו. לשם כך עלינו להוסיף לאלגוריתם המיזוג צבירה של איברים אלו. כל פעם שאיבר מהחצי התחתון של המערך מועתק ל – B, כל האיברים שכבר הועתקו מהחצי העליון אל B צריכים להיספר. צבירת ערכים אלה צבועה באדום. שאר האלגוריתם זהה למיון מיזוג.

```
MERGESORT(A,p,r)
if p<r
then q←⌊(p+r)/2⌋
     x←MERGESORT(A,p,q)
     x←x+MERGESORT(A,q+1,r)
     x←x+MERGE(A,p,q,r)
     return x
else return 0
```

```
MERGE(A,p,q,r)
i←p  j←q+1  k←1  x←0
while i≤q and j≤r do
     if A[i]<A[j]
     then B[k]←A[i]
          x←x+j-q-1
          i←i+1
     else B[k]←A[j]
          j←j+1
     k←k+1
while i≤q do
     x←x+r-q
     B[k]←A[i]
     i←i+1
     k←k+1
while j≤r do
     B[k]←A[j]
     j←j+1
     k←k+1
for i←1 to k-1 do
     A[p+i-1]←B[i]
return x
```

שאלה

Write a function in C that receives a sorted array ***a*** of length ***n*** and an integer ***x***. The function will return the number of occurrences of x in a.

For example:

a={1,1,1,5,5,7,7,7,9} x=5. The function returns 2.

a={1,1,1,5,5,7,7,7,9} x=6. The function returns 0.

The header of the function is:

int find(int a[], int n, int x)

The running time of the function is *O*(lg*n*).

פתרון

נציע אלגוריתם המבצע עיקרון דומה לעקרון החיפוש בינארי -

הלולאה הראשונה מציבה ב l את האינדקס של המופע הימני ביותר של x.

הלולאה השניה מציבה ב f את את האינדקס של המופע השמאלי ביותר של x.

```
int find(int a[], int n, int x)
{
    int p=0, r=n-1, q, l=-1, f=-1;
    while (p<=r)
    {
        q=(p+r)/2;
        if (a[q]==x)
            l=q;
        if (a[q]>x)
                r=q-1;
        else    p=q+1;
    }
    if (l==-1)
        return 0;
    p=0; r=n-1;
    while (p<=r)
    {
        q=(p+r)/2;
        if (a[q]==x)
            f=q;
        if (a[q]<x)
                p=q+1;
        else    r=q-1;
    }
    return l-f+1;
}
```

סיבוביות זמן הריצה של האלגוריתם זהה לסיבוביות זמן ריצה של חיפוש בינארי. השינויים וההוספות מתבצעים בזמן קבוע ואינם משפיעים על זמן הריצה. ועל כן זמן הריצה הכולל של האלגוריתם הוא O(lg(n))

Dynamic Programming

Question

Given a problem of using (a) rented car(s) to drive from one point to another on a highway. There are n car rental agencies along the highway. In each one you can rent a car and return it at any car agency you encounter along the way. You are not allowed to drive back in the direction you came from, which means you're required to continue in one direction only.

For every possible starting point i, and for every destination point j, the price for renting a car for that specific leg is known (and defined in matrix C), but the price of renting a car for the leg from i to j may be higher than the sum of prices required for renting various cars for shorter legs of the route. In other words, it may possible to find an alternative route, in which you rent the car at location i, return it at location k, rent a different car at location k, and return the second car at location j – and the total sum payed for both rentals, may be less than the amount payed of renting one car from location i to location j. Note: changing cars along the route does not require any additional payment. Write an algorithm that receives as it's parameter a matrix C, and prints the renting car agencies identities, on the cheapest route for location i to destination j.

נתונה בעיה של שימוש ברכב(ים) שכור(ים) לצורך נסיעה מנקודה אחת לנקודה אחרת על כביש מהיר.

ישנן *n* סוכנויות להשכרת רכב לאורך הכביש המהיר. בכל אחת מהן אפשר לשכור רכב הניתן להחזרה בכל סוכנות אחרת בהמשך הדרך.

לא ניתן לסגת לאחור (כלומר יש להמשיך באותו כיוון לאורך כל הנסיעה).

לכל נקודת מוצא אפשרית *i* ולכל נקודת הגעה אפשרית *j* , עלות השכרת רכב מ-*i* ל-*j* ידועה (ונתונה במטריצה *C*), אך יתכן כי עלות ההשכרה מ-*i* ל-*j* תהיה גבוהה מסך העלויות של השכרות "קצרות" יותר. כלומר, יתכן כי החזרת הרכב הראשון בסוכנות מסוימת *k* בין *i* ל-*j* והמשך נסיעה ברכב השני מ -*k* ל-*j* תעלה בסך-הכל פחות מאשר השכרת רכב יחיד מ-*i* ל-*j*.

החלפת רכבים במהלך הדרך אינה כרוכה בתשלום נוסף.

כתבו אלגוריתם המקבל את המטריצה *C*, ומדפיס את מספרי הסוכנויות בהן כדאי להחליף רכב.

פתרון

במטריצה a בתא [row][col] תאוחסן העלות לנסיעה מסוכנות row לסוכנות col. העלות הזאת היא המינימום בין העלויות של האפשרויות השונות: לנסוע ישירות, להחליף רכב בסוכנות row+1, להחליף רכב בסוכנות row+2, וכו'. המטריצה מתמלאת בסדר הבא (נדגים על מטריצה בגודל 5 על 5):

	1	5	8	10
		2	6	9
			3	7
				4

כלומר: קודם תחושבנה העלויות לכל הנסיעות דרך 2 סוכנויות, אחר תחושבנה העלויות לכל הנסיעות דרך 3 סוכנויות וכו'.

אחרי חישוב העלויות (בכחול), תודפסנה הסוכנויות (חוץ מהראשונה והאחרונה) בהן כדאי לשבור/להחזיר רכב.

אם a[1,dest]=a[1,i]+C[i,dest], אז בנסיעה מ-1 ל-dest ההחלפה האחרונה התבצעה בסוכנות i.

<u>hire(C)</u>

```
for len←2 to n  do
  for row←1 to n+1-len do
    col←row+len-1
    a[row][col]←C[row][col]
    for i←row+1 to col-1 do
      a[row][col]←MIN(a[row][col],a[row][i]+a[i+1][col])


dest←n
for col←n-1 downto 2 do
  if a[1][dest]=a[1][col]+C[col][dest]
  then print col
       dest←col
```

Qeustion

Given a latter with n steps. At each step it is possible to climb one or two steps.

Write a dynamic algorithm to calculate the number of possibilities to climb the latter.

For example:

for n=4 there are 5 climbing options:

1. 1 step, 1 step, 1 step, 1 step
2. 1 step, 1 step, 2 steps
3. 1 step, 2 steps, 1 step
4. 2 steps, 1 step, 1 step
5. 2 steps, 2 steps

נתון סולם בעל *n* שלבים. בכל צעד אפשר לטפס שלב אחד או שני שלבים.

כתבו אלגוריתם תכנון דינמי המחשב בכמה אפשרויות אפשר לטפס על הסולם.

לדוגמה:

עבור n=4 ישנן חמש אפשרויות טיפוס:

1. שלב, שלב, שלב, שלב
2. שלב, שלב, 2 שלבים
3. שלב, 2 שלבים, שלב
4. 2 שלבים, שלב, שלב
5. 2 שלבים, 2 שלבים

פתרון

נסמן את מספר האפשרויות לטיפוס על סולם בעל n שלבים ב: L_n. כדי לטפס על סולם בעל n שלבים אפשר להתחיל בצעד של שלב אחד ולהמשיך לטפס על n-1 השלבים הנותרים או שאפשר להתחיל בצעד של שני שלבים ולהמשיך לטפס על n-2 השלבים הנותרים.

$L_n=L_{n-1}+L_{n-2}$

האלגוריתם:

```
ladder(n)
L=1   //L_{i-2}
LL=2 //L_{i-1}
for i←3 to n do
  t←LL+L
  L←LL
  LL←t
return LL
```

שאלה

Write a function that calculates the number of n-bit sequences that do not contain a sequence of k ($1<k<n$) zeros.

For example: for n=4, k=2, there are 8 such sequences (0101, 0110, 0111, 1010, 1011, 1101, 1110, 1111).

The header of the function is:

int count(int n, int k)

The function should use a dynamic programming algorithm and its running time should be O(*nk*).

פתרון

ניתן לכתוב אלגוריתם תכנון דינמי הפותר את הבעיה בזמן O(*n*). נראה איך לעשות זאת.

נסמן את מספר רצפי הסיביות באורך *n*, שאין בהם רצף של *k* אפסים ב:

$f(n,k)$

אם $n = 0$ אז קיים רצף יחיד כמבוקש – הרצף הריק.

אם $n < k$ אז הסיבית השמאלית ביותר יכולה להיות 0 או 1, וההמשך הוא כל רצף באורך $n - 1$.
לכן: $f(n,k) = 2f(n-1,k)$ (כל רצף באורך n).

אם $n = k$ אז $f(n,k) = 2f(n-1,k) - 1$ (כל רצף חוץ מהרצף שכולו אפסים).

נניח עכשיו ש $n > k$. איך יכול להתחיל רצף כזה? יש *k* אפשרויות:

1…

01…

001…

.

.

.

$\underbrace{00\ldots0}_{k\text{-}1\ \text{אפסים}}1\ldots$

כלומר: $f(n,k)=\sum_{i=n-k}^{n-1} f(i,k)=f(n-1,k)+\sum_{i=n-1-k}^{n-2} f(i,k)-f(n-1-k,k)=$
$2f(n-1,k)-f(n-1-k,k)$

אחרי שמצאנו את המבנה הרקורסיבי נוכל לרשום פונקציה שממשת אלגוריתם לתכנון דינמי:

```
int count(int n, int k)
{
  int a[n+1];
  a[0]=1;
  for (int i=1; i<=n; i++)
  {
    a[i]=2*a[i-1]
    if (i>k)
         a[i]=a[i]-a[i-k-1];
    else if (i==k)
              a[i]=a[i]-1;
  }
  return a[n];
}
```