CS 350 Algorithms and Complexity

Winter 2019

Lecture 7: Decrease & Conquer

Andrew P. Black

Department of Computer Science

Portland State University

What is Decrease-and-Conquer?

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

decreasing n by a **constant factor**, *often* 2, or

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

- decreasing n by a **constant**, *e.g.*, 1, or
- decreasing n by a **constant factor**, *often* 2, or
- decreasing n by a variable amount

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

- decreasing n by a **constant**, *e.g.*, 1, or
- decreasing n by a **constant factor**, *often* 2, or
- decreasing n by a variable amount, *e.g.*, Euclid's algorithm

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

decreasing n by a **constant factor**, *often* 2, or

decreasing n by a variable amount, *e.g.*, Euclid's algorithm

… to get a problem instance of size $k < n$

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

decreasing n by a **constant factor**, *often* 2, or

decreasing n by a variable amount, *e.g.*, Euclid's algorithm

… to get a problem instance of size $k < n$

1. Solve the instance of size k, using the same algorithm recursively.

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

decreasing n by a **constant factor**, *often* 2, or

decreasing n by a variable amount, *e.g.*, Euclid's algorithm

… to get a problem instance of size $k < n$

1. Solve the instance of size k, using the same algorithm recursively.
2. Use that solution to arrive at the solution to the original problem.

What is Decrease-and-Conquer?

Solves a problem instance of size n by:

decreasing n by a **constant**, *e.g.*, 1, or

decreasing n by a **constant factor**, *often* 2, or

decreasing n by a variable amount, *e.g.*, Euclid's algorithm

… to get a problem instance of size $k < n$

1. Solve the instance of size k, using the same algorithm recursively.
2. Use that solution to arrive at the solution to the original problem.

Decrease-and-Conquer

- Also known as the *inductive* or *incremental* approach
- implement it <u>iteratively</u> *or* <u>recursively</u>
  - how does the iterative method work?

Decrease by a Constant: Examples

Decrease by a Constant: Examples

- Exponentiation using $a^n = a^{n-1} \times a$
- Insertion Sort
- Ferrying Soldiers
- Alternating Glasses
- Generating the Powerset

Exponentiation using $a^n = a^{n\text{-}1} \times a$

- How does the decrease-and-conquer algorithm differ from the Brute-force algorithm?

A. the decrease-and conquer algorithm is more efficient

B. the brute-force algorithm is more efficient

C. the two algorithms are identical

D. the two algorithms have the same asymptotic efficiency, but decrease-and conquer has a better constant.

Insertion Sort

- To sort array A[1..n], sort A[1..n-1] recursively and then insert A[n] in its proper place among the sorted A[1..n-1]

- Usually implemented bottom up (non-recursively)

Example: Sort 6, 5, 3, 1, 8, 7, 2, 4

6 5 3 1 8 7 2 4

Credit: Wikimedia Commons http://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif

Insertion Sort

- To sort array A[1..n], sort A[1..n-1] recursively and then insert A[n] in its proper place among the sorted A[1..n-1]

- Usually implemented bottom up (non-recursively)

Example: Sort 6, 5, 3, 1, 8, 7, 2, 4

6 5 3 1 8 7 2 4

Credit: Wikimedia Commons http://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif

Insertion Sort

```
insertionSort
    "Sort me using insertion sort.

    | v n j A |
    A ← self.
    n ← self size.
    2 to: n do: [ :i |
        v ← A at: i.
        j ← i.
        [ (j > 1) and: [ (A at: j-1) > v ] ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ]
```

Insertion Sort

```
insertionSort
    "Sort me using insertion sort. Levitin §4.1"

    | v n j A |
    A ← self.
    n ← self size.
    2 to: n do: [ :i |
        v ← A at: i.
        j ← i.
        [ (j > 1) and: [ (A at: j-1) gt: v ] ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ]
```

Insertion Sort

```
insertionSort
    "Sort me using insertion sort. Levitin §4.1"

    | v n j A |
    A ← self.
    n ← self size.
    2 to: n do: [ :i |
        v ← A at: i.
        j ← i.
        [ (j > 1) and: [ (A at: j-1) gt: v ] ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ]
```

```
gt: aNumber
    ComparisonCount ← ComparisonCount + 1.
    ↑ self > aNumber
```

recursive Insertion Sort

```
insertionSortRecursive
    "Sort me using insertion sort, using recursion rather than iteration"

    self insertionSortFirst: (self size).
    ↑ self

insertionSortFirst: n
    "Perform insertion sort on my first n elements"

    | v j |
    (n < 2) ifTrue: [ ↑ self ].
    self insertionSortFirst: (n-1).
    v ← self at: n.
    j ← n.
    [ (j > 1) and: [ (self at: j-1) gt: v]]
        whileTrue: [
            self at: j put: (self at: j-1).
            j ← j - 1 ].
    self at: j put: v.
    ↑ self
```

Analysis of Insertion Sort

Analysis of Insertion Sort

- Time efficiency

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n)$ A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n \lg n)$ D. something else

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

  $C_{avg}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n)$ A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n \lg n)$ D. something else

  $C_{avg}(n)$ A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n \lg n)$ D. something else

  $C_{best}(n) =$ A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n \lg n)$ D. something else

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

  $C_{avg}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

  $C_{best}(n)$ = A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

- Space efficiency (in addition to input):

  A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

  $C_{avg}(n)$ A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

  $C_{best}(n)$ = A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

- Space efficiency (in addition to input):

  A. Θ(n) B. Θ(n^2) C. Θ(n lg n) D. something else

- Stability: ?

- Which is the best of the following sorting algorithms?

  A. Selection Sort
  B. Bubble Sort
  C. Insertion Sort

Analysis of Insertion Sort

Analysis of Insertion Sort

- Time efficiency

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost-sorted arrays)

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost-sorted arrays)

- Space efficiency (in addition to input): $\Theta(1)$

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost-sorted arrays)

- Space efficiency (in addition to input): $\Theta(1)$

- Stability: Stable

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost-sorted arrays)

- Space efficiency (in addition to input): $\Theta(1)$

- Stability: Stable

- Insertion sort is the best elementary sorting algorithm overall

Insertion Sort with Sentinel

```
sentinalInsertionSort
    "Sort me using insertion sort, using a sentinal instead of a bounds check.

    | v n j A |
    A ← self.
    n ← self size.
    A addFirst: -100000.
    3 to: n+1 do: [ :i |
        v ← A at: i.
        j ← i.
        [ (A at: j-1) > v ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ].
    A removeFirst
```

```
insertionSort
    "Sort me using insertion sort.

    | v n j A |
    A ← self.
    n ← self size.
    2 to: n do: [ :i |
        v ← A at: i.
        j ← i.
        [ (j > 1) and: [ (A at: j-1) > v ] ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ]
```

Wirth’s Insertion Sort

```
wirthsInsertionSort
    "Sort me using N.Wirth's version of insertion sort, using an internal sentinel instead
    of a bounds check.  H. Thimbleby, Software P&E Vol 19 Nr 3, pp303-307, March 1989"

    | v n j A |
    A ← self.
    n ← self size.
    A addFirst: nil.   "make room for sentinal"
    3 to: n+1 do: [ :i |
        v ← A at: i.
        A at: 1 put: v.
        j ← i.
        [ (A at: j-1) > v ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ].
    A removeFirst
```

```
sentinalInsertionSort
    "Sort me using insertion sort, us

    | v n j A |
    A ← self.
    n ← self size.
    A addFirst: -100000.
    3 to: n+1 do: [ :i |
        v ← A at: i.
        j ← i.
        [ (A at: j-1) > v ]
            whileTrue: [
                A at: j put: (A at: j-1).
                j ← j - 1 ].
        A at: j put: v ].
    A removeFirst
```

Moral of the Story …

Moral of the Story …

- Asymptotic efficiencies don't tell the whole story!

Moral of the Story …

- Asymptotic efficiencies don't tell the whole story!
- Getting an expensive operation out of a loop can make a real-life difference

Moral of the Story …

- Asymptotic efficiencies don't tell the whole story!
- Getting an expensive operation out of a loop can make a real-life difference
- You have to measure to find out

On my Pharo System:

```
testInsertionSorts
    | anArray0 anArrayI anArrayS anArrayW n |
    n ← 10000.
    anArray0 ← (1 to: n) asOrderedCollection shuffled.
    anArrayI ← anArray0 copy.
    anArrayS ← anArray0 copy.
    anArrayW ← anArray0 copy.
    Transcript show: 'Insertion Sort: '; show: (Time millisecondsToRun: [anArray0 insertionSort]); cr.
    Transcript show: 'Rec Insertion Sort: '; show: (Time millisecondsToRun: [anArrayI  insertionSortRecursive ]); cr.
    Transcript show: 'Sentinal Insertion Sort: '; show: (Time millisecondsToRun: [anArrayS sentinelInsertionSort]); cr.
    Transcript show: 'Wirth''s Insertion Sort: '; show: (Time millisecondsToRun: [anArrayW wirthsInsertionSort]); cr.
```

On my Pharo System:

```
testInsertionSorts
    | anArray0 anArrayI anArrayS anArrayW n |
    n ← 10000.
    anArray0 ← (1 to: n) asOrderedCollection shuffled.
    anArrayI ← anArray0 copy.
    anArrayS ← anArray0 copy.
    anArrayW ← anArray0 copy.
    Transcript show: 'Insertion Sort: '; show: (Time millisecondsToRun: [anArray0 insertionSort]); cr.
    Transcript show: 'Rec Insertion Sort: '; show: (Time millisecondsToRun: [anArrayI  insertionSortRecursive ]); cr.
    Transcript show: 'Sentinal Insertion Sort: '; show: (Time millisecondsToRun: [anArrayS sentinelInsertionSort]); cr.
    Transcript show: 'Wirth''s Insertion Sort: '; show: (Time millisecondsToRun: [anArrayW wirthsInsertionSort]); cr.
```

Insertion Sort	2481	2361	2644	2290
Recursive Insertion Sort	2364	2413	2569	2258
Sentinel Insertion Sort	2187	2347	2088	1944
Wirth's Insertion Sort	2348	2527	2245	2219

Ferrying Soldiers

- A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, that it can hold just two boys or one soldier.
  - How can the soldiers get across the river and leave the boys in joint possession of the boat?
  - How many times need the boat pass from shore to shore?

Ferrying Soldiers

Ferrying Soldiers

- Apply decrease-by-1 process:

Ferrying Soldiers

- Apply decrease-by-1 process:
  - Ferry one soldier to the far side, leaving boat and boys back at their initial positions

Ferrying Soldiers

- Apply decrease-by-1 process:
  - Ferry one soldier to the far side, leaving boat and boys back at their initial positions
  - If no soldiers remain, we have finished,

Ferrying Soldiers

- Apply decrease-by-1 process:
  - Ferry one soldier to the far side, leaving boat and boys back at their initial positions
  - If no soldiers remain, we have finished,
  - otherwise, ferry remaining $n-1$ soldiers

Ferrying Soldiers

- Apply decrease-by-1 process:
  - Ferry one soldier to the far side, leaving boat and boys back at their initial positions
  - If no soldiers remain, we have finished,
  - otherwise, ferry remaining $n-1$ soldiers
- How many (one way) boat trips will it take to ferry **one** soldier?

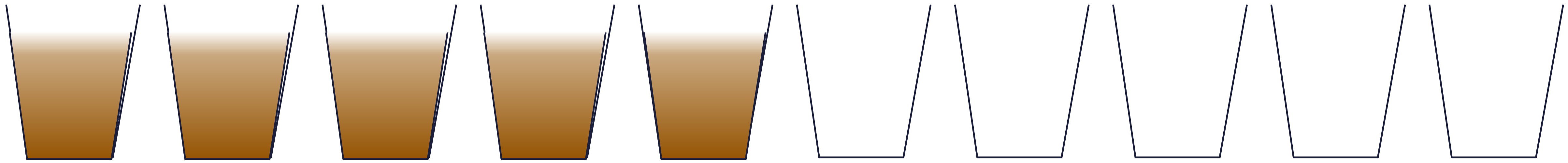
Ferrying Soldiers

- Apply decrease-by-1 process:
  - Ferry one soldier to the far side, leaving boat and boys back at their initial positions
  - If no soldiers remain, we have finished,
  - otherwise, ferry remaining $n-1$ soldiers
- How many (one way) boat trips will it take to ferry **one** soldier?

  A. 1 B. 2 C. 3 D. 4 E. 5 F. 6

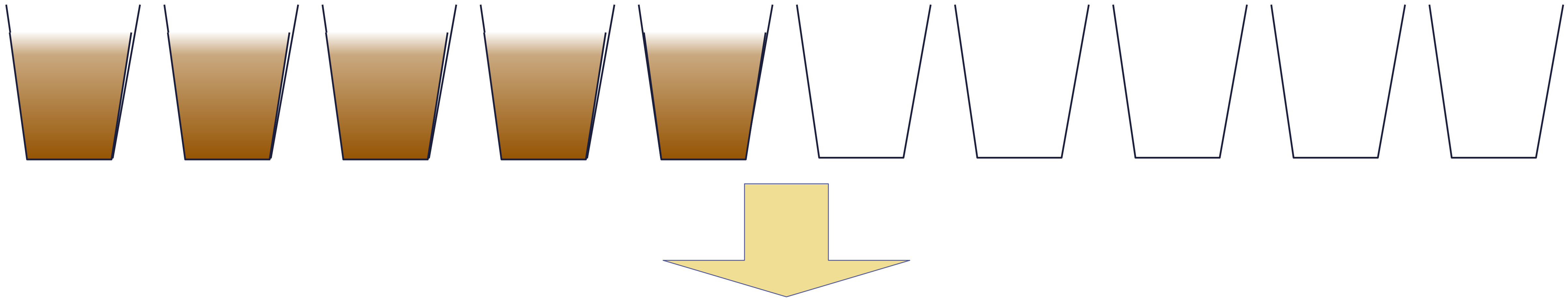
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move



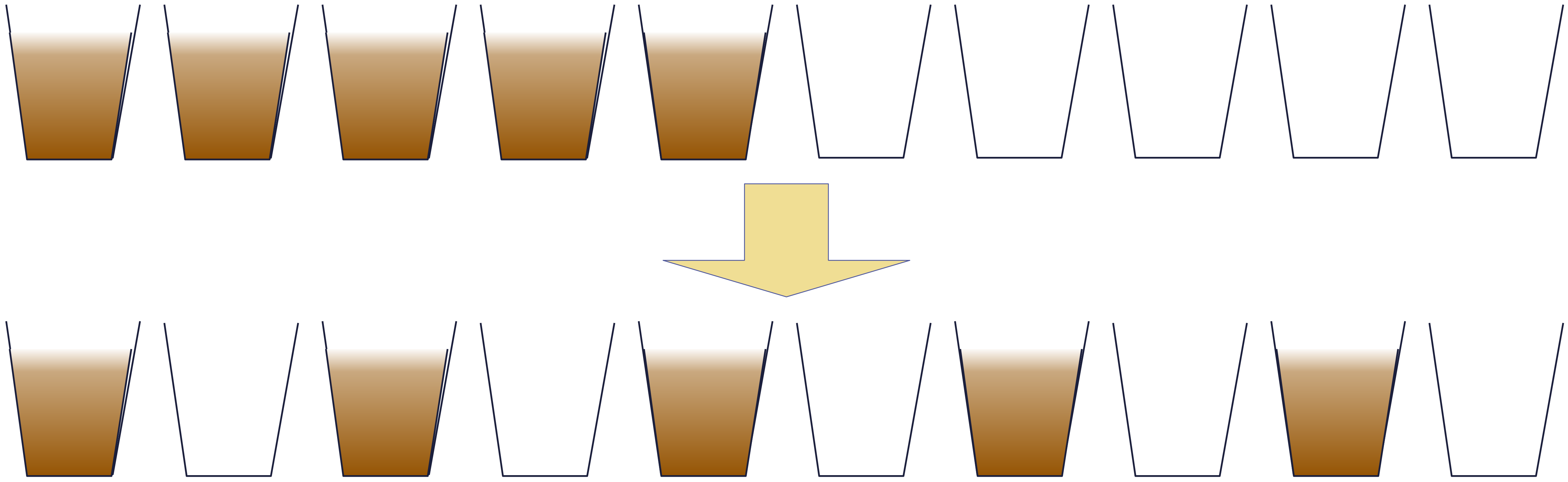
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move



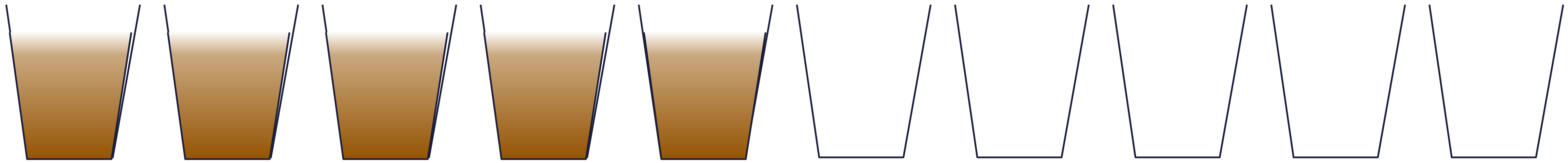
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move



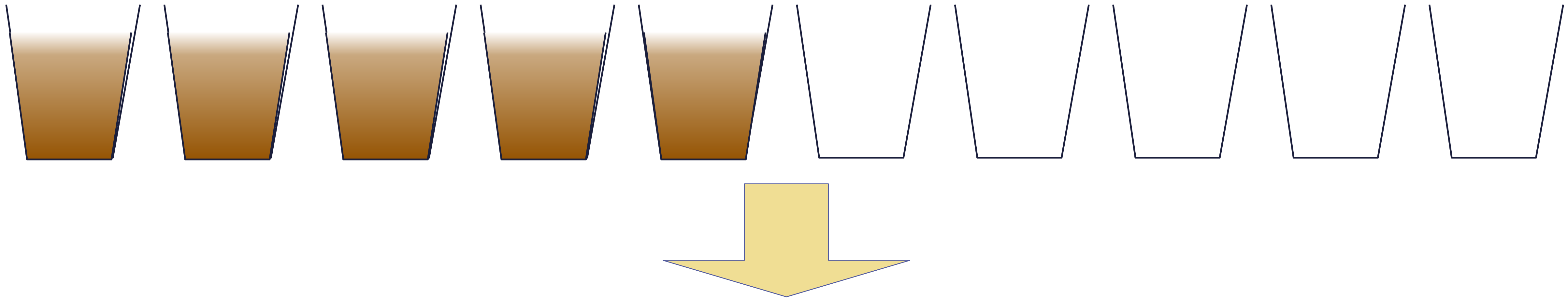
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move
- Apply decrease by-a-constant:
  - What smaller problem can we solve that will help?



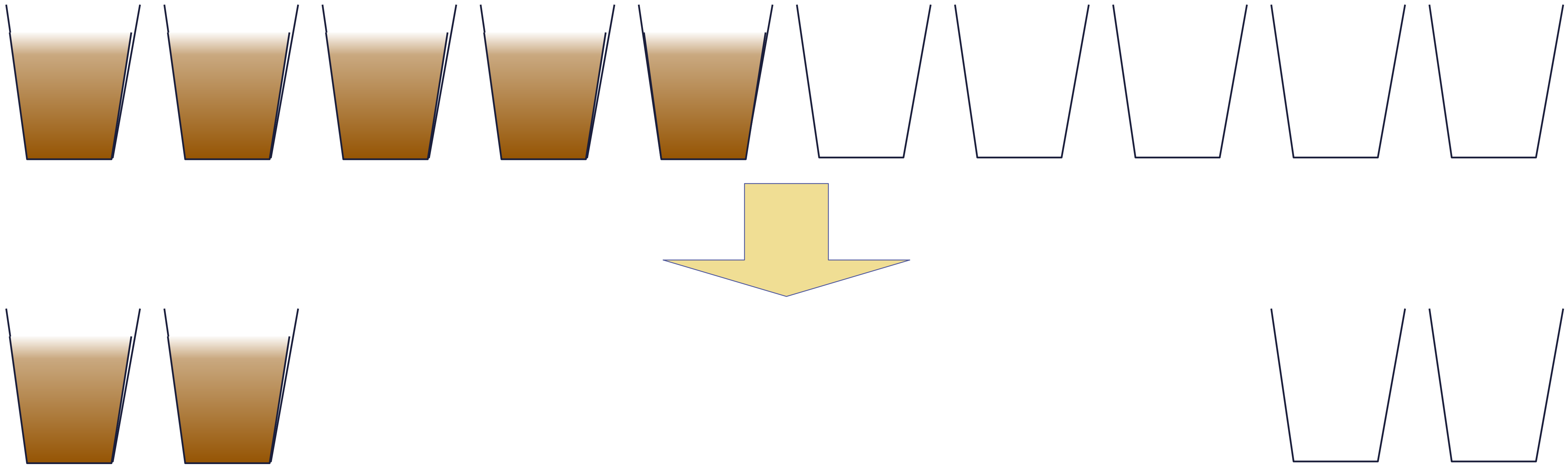
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move
- Apply decrease by-a-constant:
  - What smaller problem can we solve that will help?



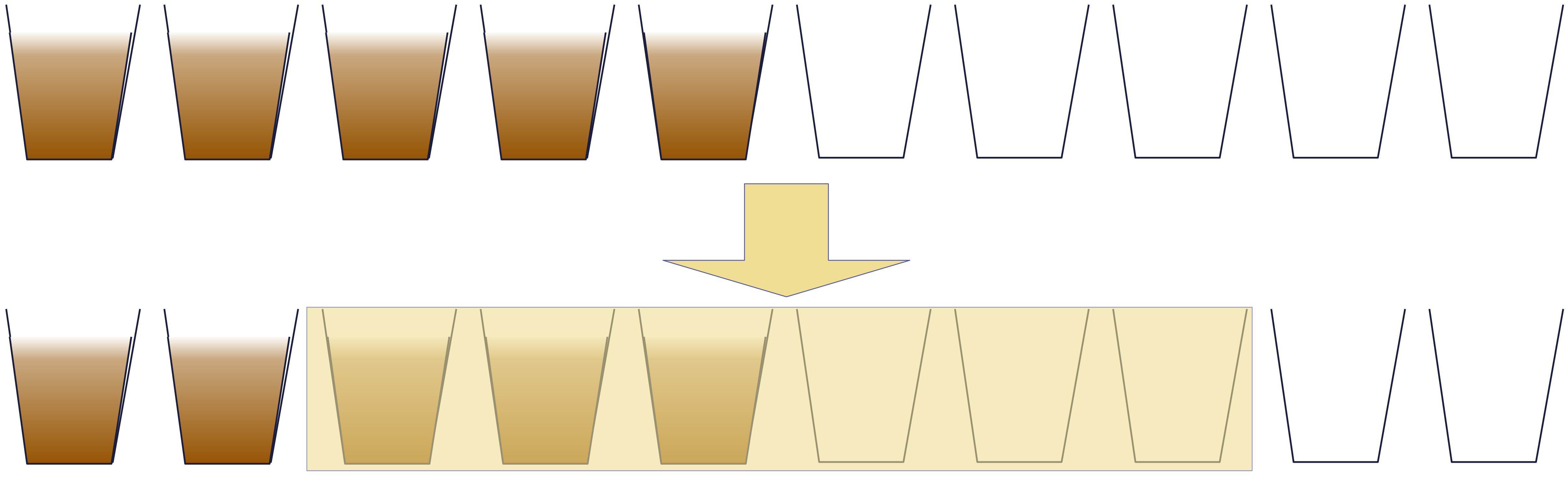
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move
- Apply decrease by-a-constant:
  - What smaller problem can we solve that will help?



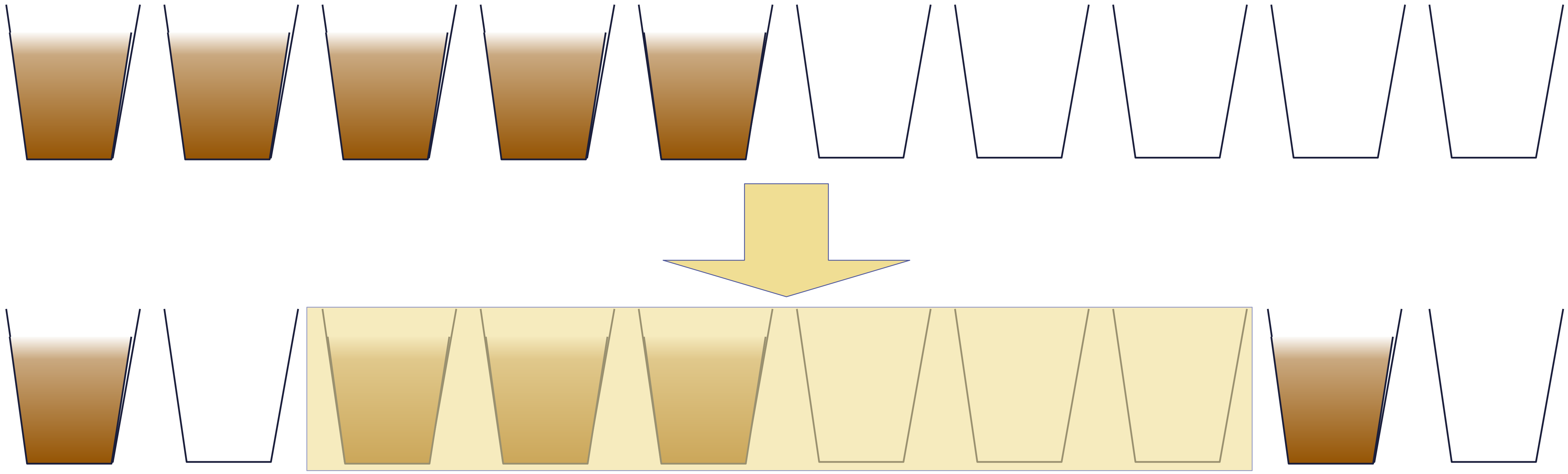
Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging two glasses is one move
- Apply decrease by-a-constant:
  - What smaller problem can we solve that will help?



Alternating Glasses

- There are $2n$ glasses standing in a row, the first n of them filled with beer, while the remaining n glasses are empty. Make the glasses alternate in a filled-empty-filled-empty pattern in the minimum number of moves.
  - Interchanging (any) two glasses is one move
- Apply decrease by-a-constant:
  - What smaller problem can we solve that will help?



Depth-first Search

- Levitin says: Depth-first Search uses a Stack, Breadth-first search uses a queue

ALGORITHM *DFS(G)*

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

ALGORITHM *BFS(G)*

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
      bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Depth-first Search

- Levitin says: Depth-first Search uses a Stack, Breadth-first search uses a queue

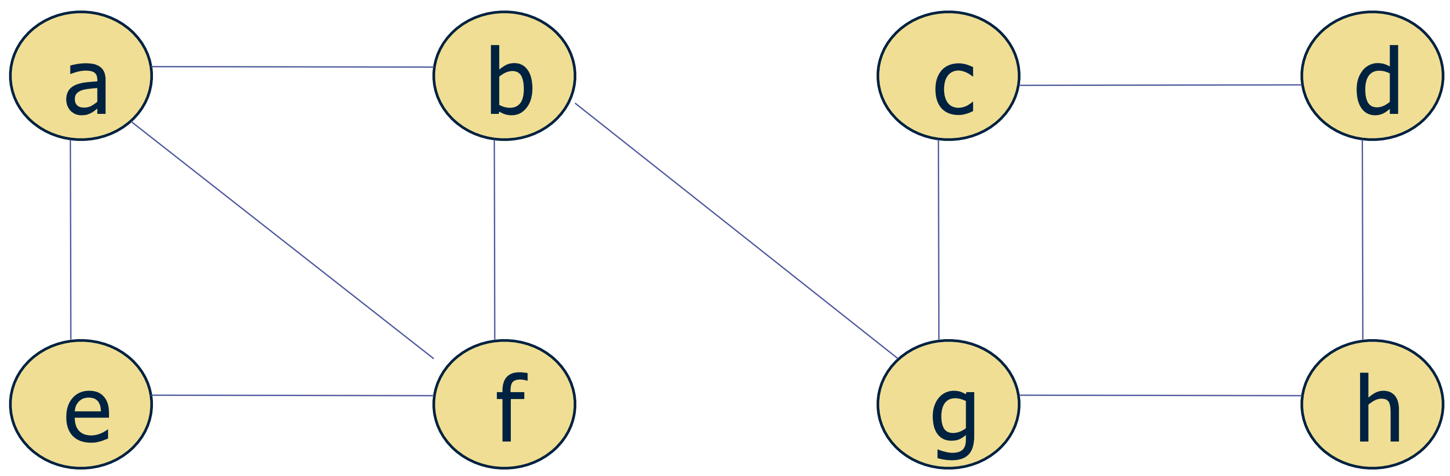
- Where's the stack?

DFS with explicit stack

- *dfs*(r)

```
S ← Stack.empty
S.push r
{S.notEmpty} whileTrue {
        u ← S pop
        u hasBeenVisited ifFalse {
                u markVisited
                u adjacentVerticesDo { v → S.push v }}}
```

Example: DFS traversal of undirected graph

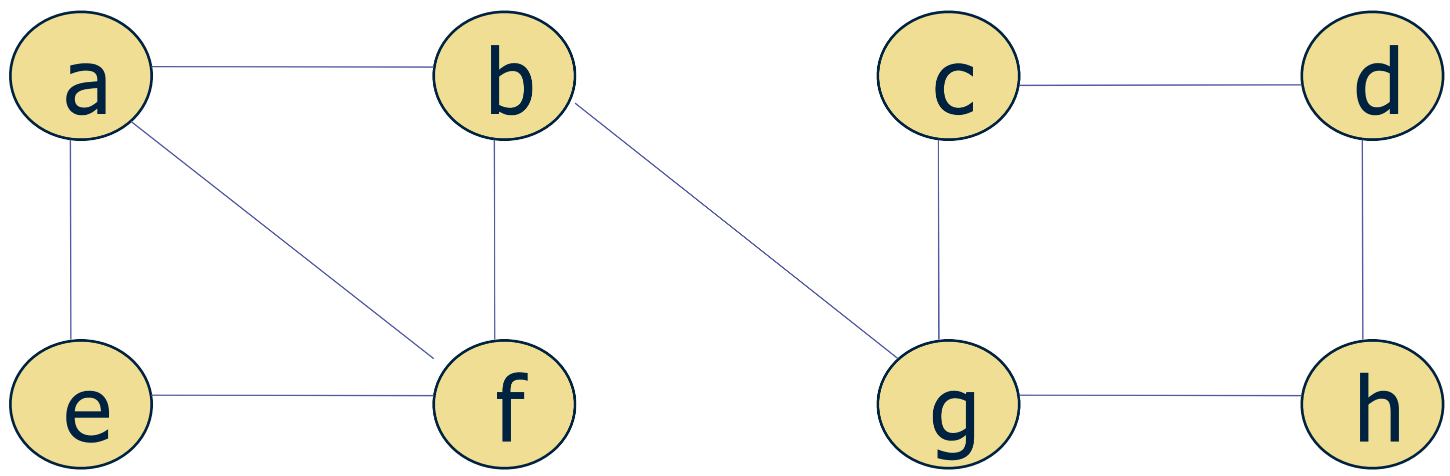


DFS traversal stack:

DFS tree:

```
dfs(r)
    S ← Stack.empty
    S.push r
    {S.notEmpty} whileTrue {
        u ← S pop
        u hasBeenVisited ifFalse {
                u markVisited
                u adjacentVerticesDo { v → S.push v }}}
```

Example: DFS traversal of undirected graph



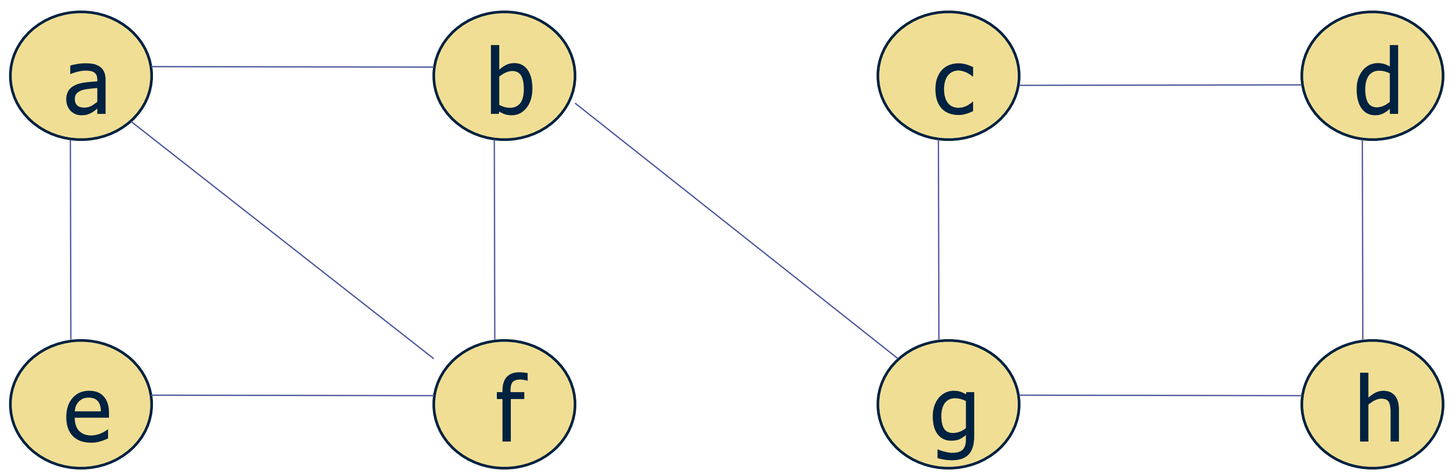
DFS traversal stack:

a

DFS tree:

```
dfs(r)
    S ← Stack.empty
    S.push r
    {S.notEmpty} whileTrue {
        u ← S pop
        u hasBeenVisited ifFalse {
                u markVisited
                u adjacentVerticesDo { v → S.push v }}}
```

Example: BFS traversal of undirected graph

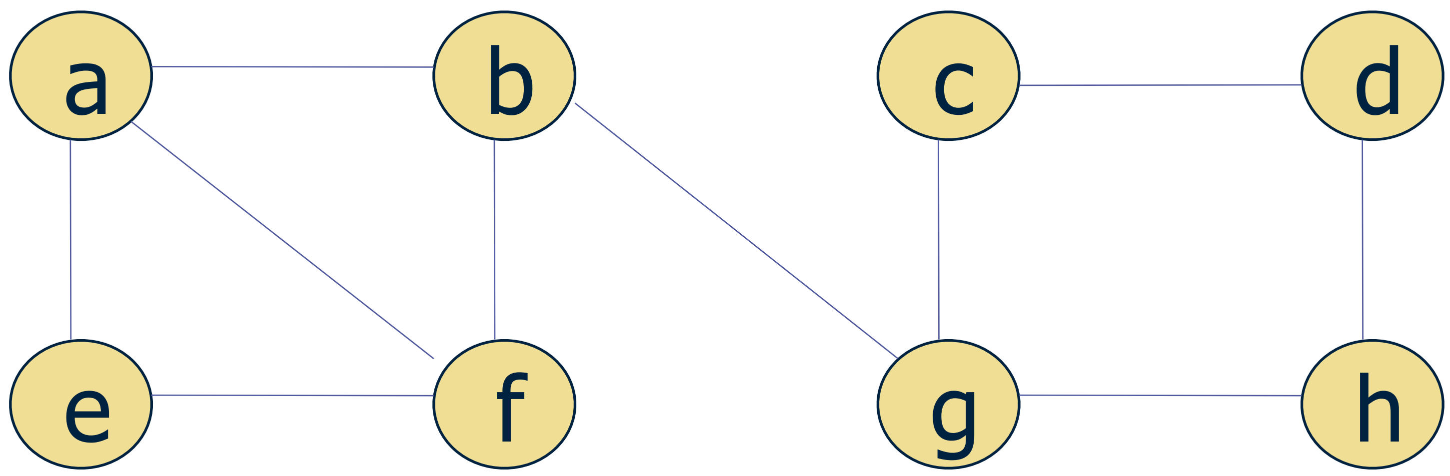


BFS traversal queue:

BFS tree:

```
bfs(r)
    Q ← Queue.empty; count ← 0
    G.allVerticesDo { v → v.markNotVisited }
    Q.add r
    {Q.notEmpty} whileTrue {
        f ← Q.remove
        f.adjacentVerticesDo { a →
            if (a.isNotVisited) then { a.markWith(count++) }
            Q.add(a) }
    }
```

Example: BFS traversal of undirected graph



BFS traversal queue:

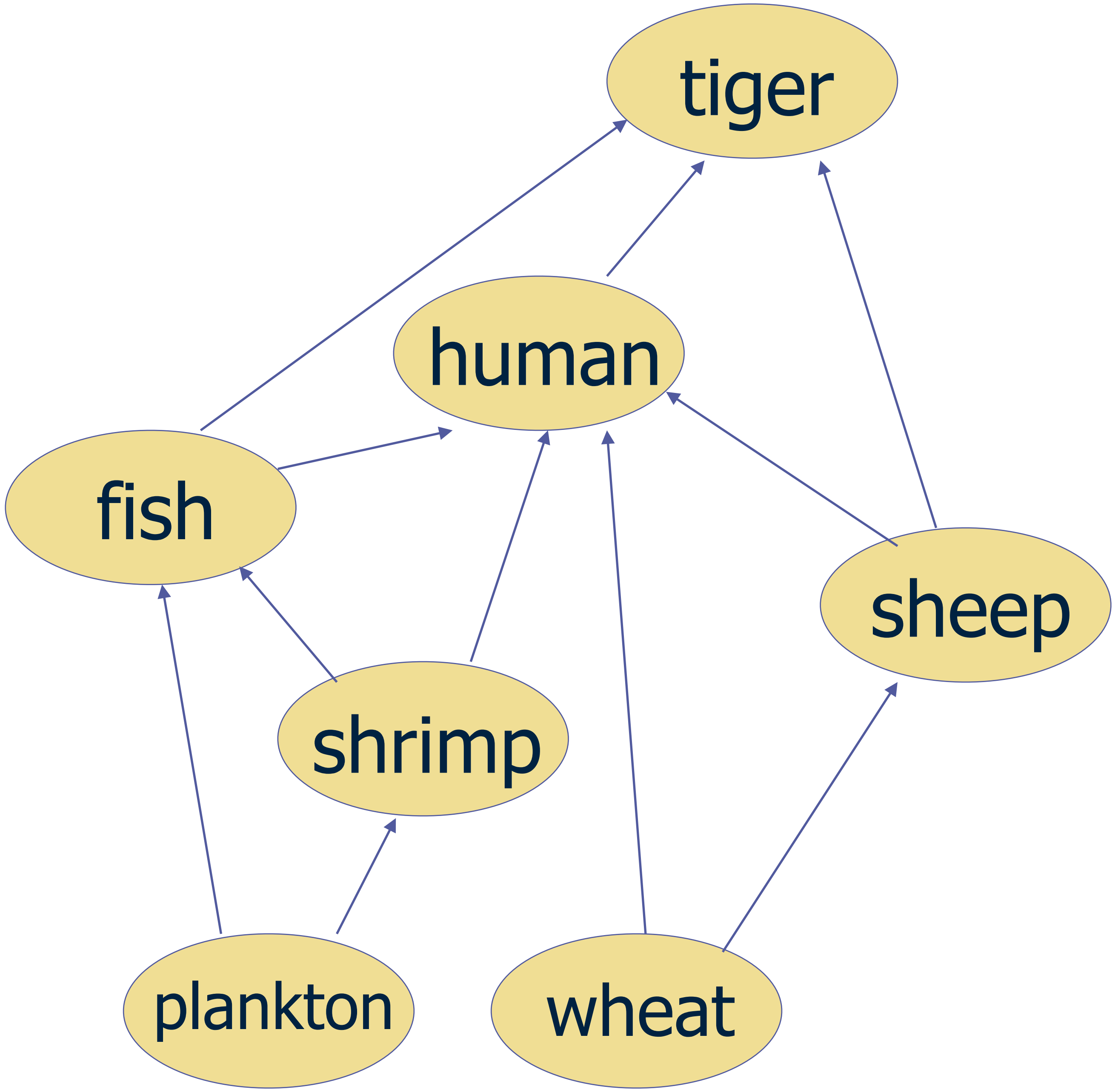
a

BFS tree:

```
bfs(r)
    Q ← Queue.empty; count ← 0
    G.allVerticesDo { v → v.markNotVisited }
    Q.add r
    {Q.notEmpty} whileTrue {
        f ← Q.remove
        f.adjacentVerticesDo { a →
            if (a.isNotVisited) then { a.markWith(count++) }
            Q.add(a) }
    }
```

Topological Sorting Example

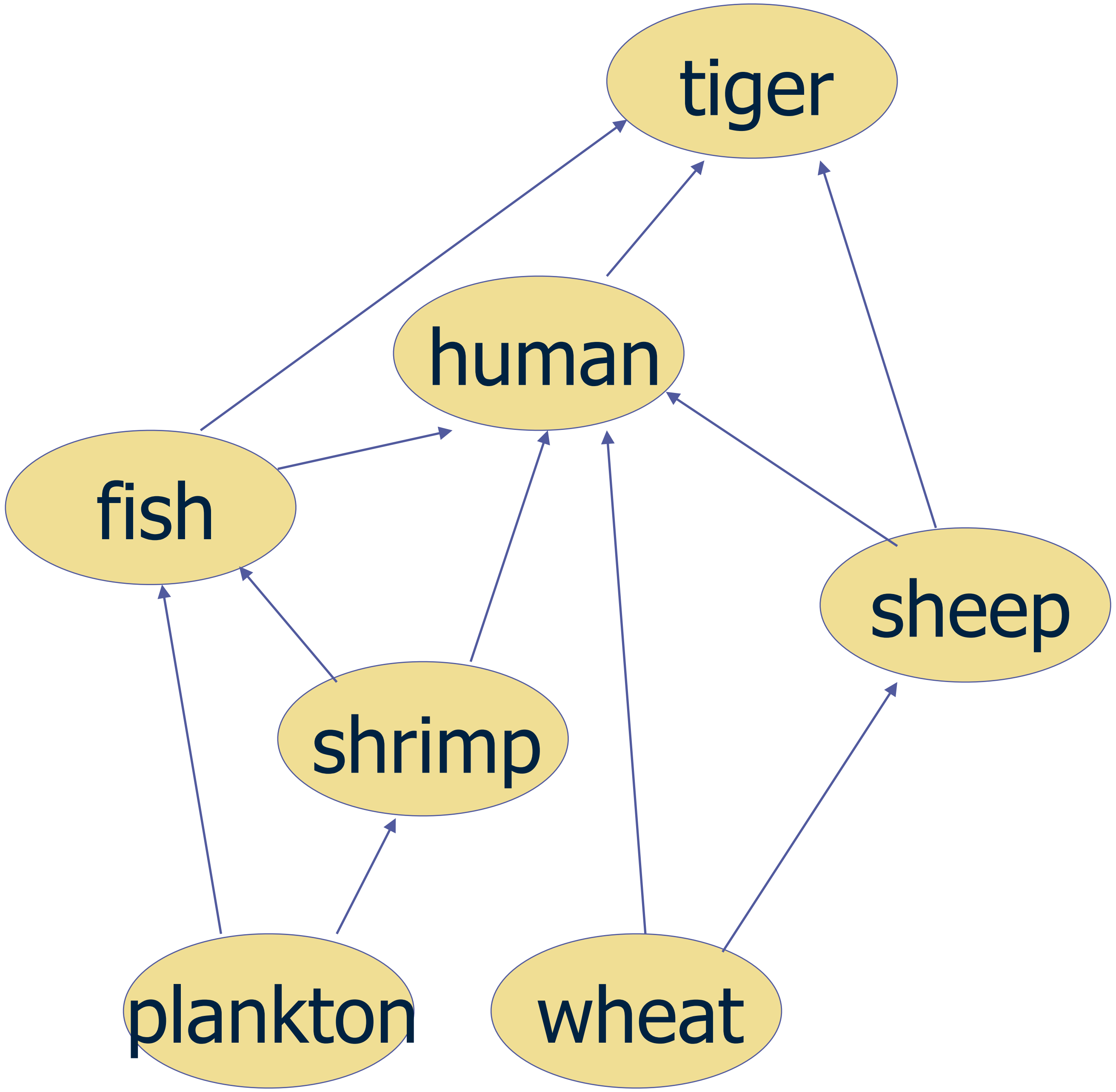
Order the following items in a food chain



Topological Sort using decrease-by-one

- Basic idea:
  - topsort a graph with one less vertex
  - combine the additional vertex with the sorted graph
- Problem:
  - How to choose a vertex that can be easily re-combined?

Which vertex should we remove?



A. fish
B. shrimp
C. plankton
D. wheat
E. sheep
F. human
G. tiger

Decrease by a Constant Factor

- binary search and bisection method (§12.4)
- exponentiation by squaring
- multiplication à la russe

Variable-size decrease

- Euclid's algorithm
- selection by partition
- Nim-like games