# Quizzes

# Intro (E1)

### Intro 1

Assume we must implement a given specification, with a precondition and a postcondition. For which input is our implementation required to establish the postcondition?
for all input
**for input that satisfies the precondition**
for no input

Which is **NOT** a valid way to express that an array B[1..n] is "non-decreasing"?
forall i forall j (1 <= i <= j <= n => B[i] <= B[j])
forall i forall j (1 <= i < j <= n => B[i] <= B[j])
**forall i forall j (1 <= i < j <= n => B[i] < B[j])**

Which is **NOT** a valid way to express that an array B[1..n] is "non-decreasing"?
**forall k (1 <= k <= n => B[k] <= B[k+1])**
forall i (1 <= i < n => B[i] <= B[i+1])
forall j (1 < j <= n => B[j-1] <= B[j])

Consider the array B with content [21,17,21,28,17].
Given the specification of the selection problem, what is then the
2nd smallest element of B
**17**
3rd smallest element of B
**21**

### Intro 2

How do we in our pseudocode express the assignment of value 7 to variable w?
w := 7
**w <- 7**
w = 7
w <-> 7

Assume that x is 5 and y is 7 before the parallel assignment
x,y <- y, x+y
What are the values of x and y afterwards?
x is
**7**
y is
**12**

Consider the iterative implementation of Insertion Sort.
It runs in time proportional to
the size of the input
**when the input is (almost) sorted**
the square of the size of the input
**when the input is randomly generated**
the cube of the size of the input
**never**

Which claims are true about the space use of the iterative implementation of Insertion Sort?
**it is in-place**
**it uses space at most logarithmic in the size of the input**
running it causes the stack to grow
it uses space proportional to the size of the input array

Let B[1..n] be an array of integers. To express that no integer occurs twice in B, we may write (check all that applies)
forall i in 1..n, forall j in 1..n, B[i] = B[j] implies i = j
**forall i in 1..n, forall j in i+1..n, B[i] != B[j]**
forall i in 1..n-1, B[i] != B[i+1]
forall i in 1..n, forall j in 1..n, i != j and B[i] != B[j]
**forall i in 1..n, forall j in 1..n, i != j implies B[i] != B[j]**
forall i in 1..n, forall j in 1..n, B[i] != B[j]

# Asymp (E1)

### Asymp 1

Assume that we have analyzed an algorithm, and found that on input of size n it runs in time (measured in microseconds)
5 n^3 + 973 n^2 + 28n + 46.
Which parts of that formula shall we consider essential information?
5
973 n^2
**n^3**
all of it

Given a set with n elements, how many subsets are there?
n^2
n^n

**2^n**
n

Find the smallest integer n such that 2^n is at least one billion (that is, 10^9)
**30**

Which functions belong to O(n^2)? (check all that apply)
**7n + 3**
2^n
**5n^2 + 4n + 8**
n^3 - 4n + 8

What is the (sufficient and necessary) condition for n^p to belong to O(n^q)?
**p <= q**
p > q
p < q
p = q
p >= q

### Asymp 2

Which functions belong to Omega(n^2)?
n lg(n)
**(n+7)^2**
3n
**5n^3 + 4n**

Which functions belong to Theta(n^2)?
3n + 8
2n^3 + 4n + 8
**5n^2 + 3n + 7**
**(n+7)^2**

Which functions belong to o(n^2)?
0.03 n^2
(n+1)^2
**n lg(n)**
**27n + 33**

Which functions belong to omega(n^2)?
n lg(n)
**n^2 lg(n)**
237 n^2 + 8n
**(1,001)^n**

The running time of insertion sort is in
O(n)
**for the best-case interpretation**
O(n^2)
**for the worst-case interpretation**
O(n^3)
**for both interpretations**
Omega(n)
**for both interpretations**
Omega(n^2)
**for the worst-case interpretation**
Omega(n^3)
**for no interpretation**
Theta(n)
**for the best-case interpretation**
Theta(n^2)
**for the worst-case interpretation**

Match each ??? with the appropriate symbol (here lg is the binary logarithm, and ln the natural logarithm)
lg(n) in ???(sqrt(n))
**little-o**
(1.001)^n in ???(n^7)
**little-omega**
lg(n) in ???(ln n)
**big-Theta**

# Loops (E1)

### Loops 1

Consider the program (with E some expression)
z <- 0
k <- n
while k > 1
z <- z+1
k <- E
For each E, state what is the asymptotic running time of the resulting program, as a function of n
E subtracts by two: k - 2
**Theta(n)**

E divides by two: k / 2
**Theta(lg n)**

Consider the program
x <- 0
m <- n *n*
*for i <- 1 to m*
*q* <- *i* i * i
for k <- 1 to q
x <- x + 1
Its asymptotic running time, as a function of n, is given by
sum_{i = 1}^{n^2 } i^3
Answer 1:
**n^2**
Answer 2:
**i^3**

Consider the program
x <- 0
for i <- 1 to n
q <- 1
while q <= i
q <- q + q
x <- x + 1
Its asymptotic running time, as a function of n, is given by
sum_{i = 1}^{n} E where E is
sqrt(i)
i
**lg i**
i^2

### Loops 2

Which functions (of n) are smooth?
1/n
2^n
**lg(n)**
**sqrt(n)**
**n^2**

Which is a correct approximation of
sum_{i = 1}^{n} 2^i
Theta(n 2^n)
**Theta(2^n)**
Theta(n^2)

According to our results,
sum_{i=1}^{n^4} sqrt(i)
is in
Theta(n^ **4** sqrt(n^ **4** )) = Theta(n^ **6** )

Consider the expression Theta(n^2 lg(n^2)). It
**can be simplified to Theta(n^2 lg(n))**
can be simplified to Theta(n^2)
can be simplified to Theta(n^2 lg(n)^2)
cannot be simplified

## Graphs (E1)

### Graphs 1

Assume that a directed graph G has 5 nodes. What is the highest possible number of edges in G?
**25**

Assume that an undirected graph G has 5 nodes. What is the highest possible number of edges in G?
**10**

What is the smallest number of nodes needed to form a cycle
in a directed graph
**1**
in an undirected graph
**3**

Assume that the undirected graph G has exactly 5 nodes, and that G is connected. What is the smallest possible number of edges in G?
**4**

Assume that the directed graph G has exactly 5 nodes, and that G is strongly connected. What is the smallest possible number of edges in G?
**5**

If a graph algorithm has running time in Theta(n+a) then we can write that running time as
Theta(n) , if we know
**a in O(n)**
Theta(n^2), if we know
**a in Omega(n^2)**

Assume that a tree has 5 nodes. Which situations are then possible (check all that applies)
it has 3 edges
it has 5 edges
**it has 4 edges**

### Graphs 2

With the Adjacency Matrix representation of graphs, which operations always run in constant time? (check all that apply)
**Put**
**Get**
**Delete**
AllFrom

With the Adjacency Lists representation of graphs, which operations always run in constant time? (check all that apply)
**AllFrom**
Get
Delete
**Put**

In which situation
can we expect the 'get' operator to take the most time?
**a dense graph represented by adjacency lists**
can we expect a graph to require the least amount of space?
**a sparse graph represented by adjacency lists**
does the AllFrom operator have a running time that is linear in the number of nodes but yet will return a very short list?
**a sparse graph represented by an adjacency matrix**

## Recur (E1)

### Recur 1

Consider the algorithm
F(A[1..n]) =
if n = 1
skip
else if n = 2
A[1] <-> A[2]
else
q <- n/2
F(A[1..q])
F(A[q+1..n])
F(A[1..q])
Its running time T(n) can be described by the recurrence
T(n) = aT(n/b) + h(n) with h in Theta(n^q).
What is a, b, q?

a = **3**
b = **2**
q = **0**

Assume that we have a recurrence for T(n), and that we want to use the substitution method to find an upper approximation of the asymptotic behavior of T(n), that is, to prove that for some function f and some constant c, T(n) <= c f(n) for n big enough. What must we do?
the method will help to find both f and c
we have to guess c; then the method will help to find f
we have to guess both f and c
**we have to guess f; then the method will help to find c**

Consider the recurrence
T(n) <= 2 T(floor(n/2)) + n^2
We may use the substitution method to prove that for some c > 0, T(n) <= c n^2 for all n >= 1.
For n >= 2, we have the calculation
T(n) =(A) 2 T(floor(n/2)) + n^2
<=(B) 2 c floor(n/2)^2 + n^2
<=(C) 2 c (n/2)^2 + n^2
=(D) (c/2 + 1) n^2
<=(E) c n^2
Match each (in)equality with its justification
A
**recurrence**
B
**induction hypothesis**
C
**property of floor**
D
**arithmetic**
E
**when c >= 2**

### Recur 2

Suppose we want to prove, by induction in n, that P(n) holds for all non-negative integers n. Which approaches (check all that apply) will be sufficient to accomplish this?
**prove P(0), and then: for all non-negative integers m, assume that P(m) holds, and then prove P(m+1)**
for all non-negative integers m, assume that P(n) holds for all non-negative integers n with n > m, and then prove P(m)
**for all non-negative integers m, assume that P(n) holds for all non-negative integers n with n < m, and then prove P(m)**

for all non-negative integers m, assume that P(n) holds for all non-negative integers n with n != m, and then prove P(m)
for all non-negative integers m, assume that P(m) holds, and then prove P(m+1)

Consider the recurrence
T(n) = 2 T(n/2) + n^2.
We want to prove that for some c > 0, T(n) <= c n^2 for all n >= q. Without special assumptions on T, what is the smallest non-negative integer q for which this is possible?
**1**

Consider the recurrence
T(n) = 2 T(n/2) + n.
We want to prove that for some c > 0, T(n) <= c n lg(n) for all n >= q. Without special assumptions on T, what is the smallest non-negative integer q for which this is possible?
**2**

Consider the recurrence
T(n) = 2 T(n/2) + 1.
We want to prove that for some c > 0, T(n) <= c n for all n >= 1. When we try to prove that by induction, for which c does the inductive step go through?
**for no c**
for all c
for c >= 1 but not when c < 1
for c >= 2 but not when c < 2

### Recur 3 **

Recall that to apply the Master Theorem on a recurrence of the form
T(n) = a T(n/b) + f(n)
we need to compute r = log_b(a) .
Match recurrences with r = log_b(a)
T(n) = 8 T(n/2) + n^2
**3**
T(n) = 4 T(n/2) + 1
**2**
T(n) = 2 T(n/2) + n
**1**
T(n) = T(n/2) + n^2
**0**
T(n) = 9 T(n/3) + n^2
**2**
T(n) = T(n/3) + n
**0**
T(n) = 8 T(n/4) + n
**1.5**
T(n) = 4 T(n/4) + 1
**1**
T(n) = 2 T(n/4) + n^2
**0.5**

Match the recurrences to their solution
(we write '+Theta(n)' for '+ f(n) where f(n) \in Theta(n)', etc):
T(n) in 3 T(n/2)+Theta(n^2) => **Theta(n^2)**
T(n) in 4 T(n/2) + Theta(n) => **Theta(n^2)**
T(n) in 2 T(n/2) + Theta(n) => **Theta(n lg(n))**
T(n) in 2 T(n/2) + Theta(1) => **Theta(n)**
T(n) in T(n/2) + Theta(n) => **Theta(n)**
T(n) in T(n/2) + Theta(1) => **Theta(lg n)**

# Correct (E1)

### Correct 1

At which points must a loop invariant hold (check all that apply)
**at the end of the loop body**
**at the beginning of the loop body**
in the middle of the loop body
**just before the loop is entered for the first time**
**just after loop exit**

What must we prove about the code before a loop? (check all that apply)
that it makes the loop guard true
that it makes the loop guard false
**that it makes the loop invariant true**

What must we prove about the loop body (check all that apply)
that it makes the loop guard false
that it makes the loop guard true
**that it makes the loop invariant true**

What can we assume when analyzing the loop body (check all that apply)
that the loop guard is false in the beginning
**that the loop guard is true in the beginning**
**that the loop invariant is true in the beginning**

What can we assume just after loop exit (check all that apply)
**that the loop invariant is true**
**that the loop guard is false**
that the loop guard is true

Assume that the postcondition is of the form
P and Q
and that the loop invariant is Q. What is then a suitable loop guard?
P
P and Q
**not P**
not Q

Consider the program (where we assume n >= 0)
x,z <- n,0
while x > 0
B
where we want the loop (with yet unspecified body B) to have invariant x + z = n.
We now propose some options for B; you should evaluate each.
x,z <- x+1,z+1
**does not maintain invariant and does not make progress towards termination**
x,z <- x+1,z-1
**maintains invariant, but does not make progress towards termination**
x,z <- x-1,z+1
**maintains invariant and makes progress towards termination**
x,z <- x-1,z-1
**makes progress towards termination, but does not maintain invariant**

### Correct 2

Consider the loop invariant p>=1 and forall i with 1 <= i < r : A[i] > i
Which loop bodies will always maintain the invariant
**r <- r - 1**
**if p < r then A[p] <- A[p] + 1**

Consider the iterative Dutch National Flag algo as described in the lecture notes. According to the loop invariant, the items with color yet unknown are
**Between the red items and the white items**

Consider the iterative Dutch National Flag algo as described in the lecture notes. In which case does it not swap elements?
**when it sees a white item**

Which claims about the Dutch National Flag algo are correct?
**it is "in-place"**
**it runs in time proportional to the size of the input**

When asked why the Dutch National Flag algo is remarkable, people may come up with some reasons. Which are valid?
**an obvious solution to the problem is an algorithm that uses space proportional to the size of the input**

### Correct 3

Consider the algorithm for iterative insertion sort, where the outer loop has invariant
1 <= i <= n+1 with A[1..i-1] non-decreasing
Assuming n >= 1, which preamble will establish this invariant? (check all that applies)
i <- 3
i <- 0
**i <- 1**
**i <- 2**

Consider the algorithm for iterative insertion sort, as mentioned in the notes. The invariant for the inner loop states that A[k1] <= A[k2] for all k1, k2 with 1 <= k1 <= k2 <= i, except
when k2 = k1 + 1
when k2 > k1 + 1
**when k2 = j**
when k1 = j

When we see a function call F(y) inside the body of a function G (where F may or may not equal G), what should we use to reason about that call?
both the specification and implementation of F
**only the specification of F**
only the implementation of F

Assume we have an algorithm, taking input x and returning output r, that is supposed to implement for x with x >= 2 a specification with precondition Pre(x) and postcondition Post(x,r).
Assume that algorithm is implemented by a recursive function F(x) = ...
Assume that the body of F makes a recursive call F(y).
After that call, we would like to be able to infer Post(y,r).
To do so, what must we verify (check all that applies)
**that y is smaller than x (according to some integer measure)**
**that y >= 2**
**that before the call, Pre(y) holds**
that x is smaller than y (according to some integer measure)

## Divide Conquer (E2)

### DivConq 1

When is the main work done for
Merge Sort?
**after the recursive calls**
Quicksort?
**before the recursive calls**

The asymptotic running time of Merge Sort is in
Theta(n^2)

**Theta(n lg(n))**
Theta(n)

What is the asymptotic running time for Quicksort, assuming that the pivot is chosen as the
median
**Theta(n lg(n))**
smallest element
**Theta(n^2)**
90th percentile
**Theta(n lg(n))**

A space-efficient implementation of QuickSort uses space in
Theta(n)
**Theta(lg(n))**
Theta(n lg(n))
Theta(1)

### DivConq 2

We saw algorithms that multiply two 2n-bit integers, by doing suitable multiplications of n-bit integers. How many such multiplications will suffice?
**3**

Given an algorithm that multiplies two n-bit integers, what can we always say about its running time?
in O(n)
**in Omega(n)**
we cannot say anything

The recurrence for the naive divide & conquer algorithm to multiply two n * n matrices is
T(n) = **8**T(n/2) + Theta(n^**2**)

Is the naive divide & conquer algorithm to multiply two n * n matrices asymptotically faster than the simple iterative algorithm?
no, it is even asymptotically slower
**no, they both run in Theta(n^3)**
yes, it is asymptotically faster
no, they both run in Theta(n^2)

The recurrence for Strassen's algorithm to multiply two n * n matrices is
T(n) = **7**T(n/2) + Theta(n^**2**)

Assume we know that the matrices we multiply are sparse, in that all elements but a few are zero. What can we then say about the algorithms we have developed?
they are still applicable and one should still use them
they are no longer applicable
**they are still applicable but faster algorithms exist**

### DivConq 3

If the pivot for the selection problem is chosen in the most unfortunate way, the running time will be
Theta(n lg(n))
Theta(n)
**Theta(n^2)**

The linear time algorithm for the selection problem, developed in the course notes, uses a pivot which is
the median of the given numbers
the arithmetic average of the given numbers
**the median of the medians of certain chunks of the given numbers**
the arithmetic average of the medians of certain chunks of the given numbers

Consider the linear time algorithm for the selection problem, applied to the array A given by
20 14 22 30 19 2 30 6 28 3 29 5 7 9 32
where the first 5 elements form one chunk, the next 5 elements another chunk, etc.
Which number will be used as the pivot?
**9**

Consider the recurrence for the algorithm we develop in the course notes:
T(n) = T(n/5) + T(qn) + Theta(n)
Match each part of the algorithm with what it contributes to the recurrence:
finding the medians of each chunk of 5
**Theta(n)**
finding the median of the medians
**T(n/5)**
applying the Dutch National Flag algorithm
**Theta(n)**
doing a recursive call on the appropriate partition
**T(qn)**

## Heaps (E2)

### Heaps 1

Consider the binary heap represented as an array with elements
19 16 17 12 14
Which claims are true? (check all that apply)
**16 is a child of 19**
**12 is a child of 16**
**17 is a child of 19**
14 is a child of 17

12 is a child of 19
**14 is a child of 16**

Consider the binary heap represented as an array with elements
19 18 13 14 16
and suppose that the key of the root changes from 19 to 12:
12 18 13 14 16
To restore the heap property, we need to sift down the root. Which heap results?
18 16 12 14 13
18 13 12 14 16
18 12 13 14 16
**18 16 13 14 12**

Consider the binary heap represented as an array with elements
19 17 14 11 13
and suppose the key of the leftmost leaf changes from 11 to 18:
19 17 14 18 13
After percolating up the leaf 18, which heap results?
**19 18 14 17 13**
19 18 14 13 17
19 14 18 17 13
19 17 18 14 13
18 19 14 17 13

Assume that we use a binary heap to implement a priority queue. What can we then say about the (worst case) running times of the various operations, as a function of n (the number of nodes):
Insert
**Theta(lg n)**
FindMax
**Theta(1)**
DeleteMax
**Theta(lg n)**

### Heaps 2

Consider the algorithm presented for converting an array into a binary heap. If we apply it to the array
15 17 18 13 19
what is the first step?
swap 15 with 17
swap 18 with 19
swap 15 with 18
**swap 17 with 19**

The algorithm presented for converting an array with n elements into a binary heap has running time in Theta(**n**) and **is in-place**.

The heapsort algorithm for sorting n elements has running time in Theta(**n lg(n)**) and **is in-place**.

Consider the heapsort algorithm for sorting an array A[1..n] in non-decreasing order: we first convert A into a heap, and then execute the loop
for i <- n downto 2
A[1] <-> A[i]
sift down A[1] in the heap A[1..i-1]
At the beginning of each iteration (and at the end of the last iteration when i = 1), what is then always true?
**A[i+1..n] is non-decreasing**
A[i+1..n] is a heap
**the i'th smallest element is in A[1..i]**
**A[1..i] is a heap**
A[1..i] is non-decreasing

# Sorting (E2)

### Sorting 1 (and only)

Which of the below sorting algorithms are in-place?
**InsertionSort**
**HeapSort**
**QuickSort**
MergeSort

Which of the below sorting algorithms are guaranteed to have a worst-case running time in O(n lg(n))?
**MergeSort**
QuickSort
**HeapSort**
InsertionSort

Which of the below sorting algorithms can easily be made stable?
QuickSort
**MergeSort**
HeapSort
**InsertionSort**

What is the smallest number of yes/no questions you can ask and yet *always* successfully guess a number between 1 and 1,000,000,000?
**30**

How many lists with 5 different elements can we make from the numbers 21, 23, 25, 27, 29? (one such list is [27,21,25,29,23])
**120**

Given an array with 3 different elements, what is the smallest number of yes/no questions you can ask and yet always know which element is the smallest, and which is the largest?
**3**

Given an array with n different elements, what is the smallest number of yes/no questions you can ask and yet always know which element is the smallest, which is the second smallest, which is the third smallest, etc, etc?
**lg(n!)**
lg(n)
n
n!

Given a sorting algorithm based on comparisons. What can we *always* say about its running time?
we cannot say anything
**in Omega(n lg(n)) but not necessarily in O(n lg(n))**
in O(n lg(n)) but not necessarily in Omega(n lg(n))
in Theta(n lg(n)) # Dynamic Programming (E2)

# Union Find (E2)

### UnionFind

To implement a union-find structure, we may use a table to associate each element with its representative. The main disadvantage of doing so is that, unlike some other approaches,

this will take up space linear in the number of nodes
**the Union operator may require time linear in the number of nodes**
the Find operator may require time linear in the number of nodes

"Union-by-rank" means that the root of the shorter tree becomes a child of the root of the other tree
Answer 1:
**shorter tree**

Match implementations with the worst-case running time (as a function of n, the number of elements) of the Find operation :
Each node has a direct pointer to its representative
**Theta(1)**
Forest of trees with the root the representative, using arbitrary union
**Theta(n)**
Forest of trees with the root the representative, using union by rank
**Theta(lg(n))**

Using path compression, in addition to doing union-by-rank, the cost of doing m operations on a structure with n elements is
Theta(m lg(n))
linear: Theta(m)
**almost-linear: Theta(m alpha(n)) where alpha is for all practical purposes bounded by a constant**

# Greedy (E3)

### Greedy 1

Assume that we are given a number of events with fixed start and finish times, such that two events cannot go on at the same time. Which of the below strategies (check all that apply) are then guaranteed to accommodate as many events as possible?
Keep selecting the possible event (not conflicting with already scheduled events) which has
shortest duration
fewest overlapping events
**earliest finish time**
latest finish time
**latest start time**
earliest start time

Given a greedy (and deterministic) strategy G for scheduling jobs.
if the schedule produced by G can be improved by swapping two jobs
**then G is not optimal**
if the schedule produced by G can NOT be improved by swapping two jobs
**then G is optimal**
if all schedules NOT produced by G can be improved by swapping two jobs
**then G is optimal**
if some schedules not produced by G can be improved by swapping two jobs
**then G may or may not be optimal**

### Greedy 2

A given undirected connected graph
may have several different minimum spanning trees, even with different number of edges
has a unique minimum spanning tree
**may have several different minimum spanning trees but they all have the same number of edges**

Given two edges in a minimum spanning tree produced by one of our two algorithms, which edge has been added first if we used
Kruskal's algorithm
**the one with lowest weight**
Prim's algorithm
**we cannot tell**

For a graph represented by adjacency lists, Kruskal's algorithm runs in time (with n the number of nodes and a the number of edges)
**Theta(a lg(n))**
Theta(n^2)
Theta(n lg(n))
Theta(n+a)

For a graph represented by adjacency lists, Prim's algorithm runs in time (with n the number of nodes and a the number of edges)
with priority queues:
**Theta(a lg(n))**
without priority queues:
**Theta(n^2)**

### Greedy 3

Dijkstra's algorithm computes the shortest paths
from a given node to any node
Answer 1:
**a given node**
Answer 2:
**any node**

Does Dijkstra's algorithm allow the input graph to have edges with negative length?
yes, it will always work
yes, it works as long as there are no cycles with negative length
**no, then it may not compute the correct answers**

Assuming that graphs are represented by adjacency lists (with n the number of nodes and a the number of edges), the running times of Dijkstra's algorithms will be:
with priority queues:
**Theta(a lg(n))**
without priority queues:
**Theta(n^2)**

Below are some purported differences between Dijkstra's algorithm and Prim's algorithm. Find the one which is NOT correct.
**Dijkstra's algorithm has a better asymptotic running time than Prim's algorithm**
Dijkstra's algorithm computes the shortest distances from the source whereas Prim's algorithm computes a minimal spanning tree
Dijkstra's algorithm is for directed graphs whereas Prim's algorithm is for undirected graphs
for each node, Dijkstra's algorithm keeps track of its distance from the source whereas Prim's algorithm keeps track of its distance from the current tree

Dijkstra's algorithm solves the single-source shortest path problem, and can be implemented to run in time Theta(n^2) with n the number of nodes. Which claims are true (check all that apply).
**running Dijkstra's algorithm n times will solve the all-pairs shortest path problem in time Theta(n^3)**
**Dijkstra's algorithm can easily be adapted to solve also the single-destination shortest path problem, still running in time Theta(n^2)**
running a specialized version of Dijkstra's algorithm will solve the single-source single-destination shortest path problem in time Theta(n)

Assume that during execution of Dijkstra's algorithm, the set S contains A (the source), F, and H. Assume that d(C) = 7 and d(F) = 4.
What is then the most precise we can say about the length of
the shortest path from A to F
**exactly 4**
the shortest path from A to C
**at most 7**
the path composed by an edge from A to F followed by an edge from F to C
**at least 7**

### Greedy 4

Which of the below strategies (check all that apply) will always produce an optimal solution for the fractional knapsack problem?
Keep picking (taking as much of as possible) the item with
highest difference value - weight
lowest weight
highest value
**highest ratio value/weight**

Assume we have 3 symbols: A with frequency 0.5, B with frequency 0.2, C with frequency 0.3. Which of the below are optimal binary encodings (check all that apply):
**A: 0; B: 11; C: 10**
A: 0, B: 10; C: 1
A: 1; B: 00; C: 01
A: 1, B: 01; C: 0
A: 00, B: 1; C: 01
**A: 0; B: 10; C: 11**
A: 10; B: 11; C: 0

Consider the situation where we want to transmit a sequence of symbols from the alphabet {A,B,C,D}, with frequencies given by
A: 0.34, B: 0.17, C: 0.33, D: 0.16
In an optimal encoding (as produced by Huffman's algorithm), how many bits are used to represent
A = **1**
B = **3**
C = **2**

# DFS (E3)

### DFS 1

Running DFS on an undirected graph may produce which kind of edges (check all that apply)
forward edges
**tree edges**
cross edges
**back edges**

Assume that running DFS on an undirected graph results in a tree, and a back edge from u to w. How do u and w possibly relate in the tree? (check all that apply)
**w is the great-grandparent of u**
**w is the grandparent of u**
w is an uncle of u

w is the parent of u
w equals u

Given a connected undirected graph. Assume that DFS discovers u after node w is finished. What are possible ways that u and w relate in the resulting tree? (check all that apply)
one is the parent of the other
**they are cousins**
**they are siblings**
one is the grandparent of the other

Let G = (V,E) be a directed graph, and let T be a tree that results from running Depth-First Search (DFS) on G.
Let e in E be an edge from u to w.
Match each situation with the kind of edge that e is.
w is in T a grand child of u
**forward edge**
w is in T a child of u
**tree edge**
w is in T an uncle of u
**cross edge**
w is in T the parent of u
**back edge**

Given an undirected graph G = (V,E), not necessarily connected.
Assuming that the various actions each take constant time, what is then the asymptotic running time of the generic DFS algorithm, expressed as simply as possible?
Theta(|E|)
Theta(|V| * |E|)
Theta(|V|)
**Theta(|V| + |E|)**

### DFS 2

Assume we run DFS on a directed graph. For which kind of edges will the finish time of the target be earlier than the finish time of the source? (check all that apply)
**tree edges**
**cross edges**
**forward edges**
back edges

Given an acyclic directed graph, what can we say about the number of ways we can give it a topological sort?
there could be zero, one, or more than one
**there is always at least one, and possibly more than one**
there is at most one, and possibly zero
there is always at least two
there is always exactly one

The algorithm for topological sort lists nodes according to
**their finish time during depth-first-search**
their discovery time during depth-first search
their distance to the starting node

In the tree produced by a depth-first-search in undirected graph G, when is the root an articulation point for G?
when it has exactly one child
always
**when it has at least two children**
never

In the tree produced by a depth-first-search in undirected graph G, when is a leaf an articulation point for G?
**never**
when it does not have a back edge up in the tree
always

What is the running time, simplified as much as possible, of the algorithm we developed for finding articulation points in a connected undirected graph (V,E)?
**Theta(|E|)**
Theta(|V|)
Theta(|V| * |E|)
Theta(|V| + |E|)

# Flow (E3)

### Flow 1

Assume that F is a flow, and that an edge e has capacity 7.
What is the most precise we can say about F(e)?
**at most 7**
at least 7
greater than 7
exactly 7
less than 7

Assume that we have a flow network with 4 nodes: a, b, the source s, and the sink t.
For which nodes (check all that apply) do we know that the sum of incoming flow equals the sum of outgoing flow?
s
**a**
t
**b**

Consider the naive approach for computing flow that is sketched in the notes, which keeps subtracting from capacities but never adds capacities or edges. Why do we reject that approach?
**the flow it computes may not be maximal**

the flow it computes may not satisfy flow preservation
the flow it computes may exceed the capacities
the running time may be exponential

If several augmenting paths exist, the Ford-Fulkerson method
**could select any of them**
will select a path where the minimum capacity of the edges is highest
will select a path with the fewest number of edges

For an integer network (V,E) with M the maximal flow, Ford-Fulkerson's method may iterate
at most |V||E| times
**at most M times**
arbitrarily many times

## Flow 2

If several augmenting paths exist, the Edmonds-Karp algorithm
could select any of them
will select a path where the minimum capacity of the edges is highest
**will select a path with the fewest number of edges**

What can we say (check all that apply) about the running time of the Edmonds-Karp algorithm, for a graph (V,E) with maximum flow M?
**in O(M |E|)**
**in O(|V| |E|^2)**

Assume that a given flow network has a flow with value M, and a cut with capacity C.
Then the most precise we can say about their relationship is that M <= C.
Answer 1:
**<=**

Assume that a given flow network has maximum flow M. What is the most precise we can say about the number of cuts with capacity M?
at most one
**at least one**
zero
more than one
exactly one
zero, one or more

When we use the Ford-Fulkerson method to find a maximal matching, and a node becomes matched, it
will remain matched and with the same partner
**will remain matched but perhaps get another partner**
may become unmatched