

## DOMANDE CORSO ALGORITMI E STRUTTURE DATI, PROF. MANIEZZO

1. Con dimensione  $n$  di un problema ci riferiamo
  - alla lunghezza dell'algoritmo che lo risolve
  - al numero di istruzioni del programma che lo risolve
  - alla dimensione dell'input dell'istanza in considerazione
  - al numero di cicli nell'algoritmo che lo risolve
  - alla dimensione delle strutture dati utilizzate per risolverlo
2. L'analisi asintotica serve per ottenere una stima del tempo di esecuzione
  - per input molto grandi, considerando i dettagli implementativi
  - per ogni input, considerando i dettagli implementativi
  - per input molto grandi, astraendo i dettagli
  - per input molto piccoli, astraendo i dettagli
  - per input molto piccoli, considerando i dettagli implementativi
3. Supponiamo che  $f(n)$  sia  $O(g(n))$ . Allora
  - $g(n)$  non è  $O(f(n))$
  - $g(n)$  è  $O(f(n))$
  - $g(n)$  può essere  $O(f(n))$
  - $g(n)$  può essere  $o(f(n))$
  - $g(n)$  può essere  $\Omega(f(n))$
4. Supponiamo che  $f(n)$  sia  $o(g(n))$ . Allora
  - $g(n)$  non è  $O(f(n))$
  - $g(n)$  è  $O(f(n))$
  - $g(n)$  può essere  $O(f(n))$
  - $g(n)$  può essere  $o(f(n))$
  - $g(n)$  può essere  $\Omega(f(n))$
5. Supponiamo che  $f(n)$  sia  $\Theta(g(n))$ . Allora
  - $g(n)$  non è  $O(f(n))$
  - $g(n)$  è  $O(f(n))$
  - $g(n)$  può essere  $O(f(n))$
  - $g(n)$  è  $o(f(n))$
  - $g(n)$  è  $\Omega(f(n))$
6. La notazione O-grande si usa
  - nella valutazione del costo computazionale di un algoritmo nel caso ottimo
  - nella valutazione del costo computazionale di un algoritmo nel caso medio
  - nella valutazione del costo computazionale di un algoritmo nel caso pessimo
  - per determinare limiti inferiori di complessità di problemi computazionali

7. La notazione  $\Omega$ -grande si usa
- nella valutazione del costo computazionale di un algoritmo nel caso ottimo
  - nella valutazione del costo computazionale di un algoritmo nel caso medio
  - nella valutazione del costo computazionale di un algoritmo nel caso pessimo
  - per determinare limiti inferiori di complessità di problemi computazionali
8.  $0.2 n \log n + 200n + n^{1/2}$  è
- $O(n)$
  - $O(n^2)$
  - $O(n \log n)$
  - $o(n^2)$
  - $o(n)$
9.  $n^2 + n/3 + n \log n$  è
- $O(n)$
  - $O(n^2)$
  - $O(n \log n)$
  - $o(n^2)$
  - $o(n)$
10.  $n(n-1)/2$  è
- $O(n)$
  - $O(n^2)$
  - $O(n \log(n))$
  - $o(n^3)$
  - $o(n)$
11.  $20 \log n + 36 n^2 \log(n)$  è
- $O(n)$
  - $\Omega(n^2)$
  - $O(n \log(n))$
  - $o(n^2)$
  - $o(n)$
12.  $20n + 36 n \log(n)$  è
- $\Omega(n)$
  - $\Omega(n^2)$
  - $\Omega(n \log(n))$
  - $\Theta(n^2)$
  - $\Theta(n)$

13.  $n^2 + n/3 + n \log(n)$  è

- $\Omega(n)$
- $\Omega(n^2)$
- $\Omega(n \log(n))$
- $\Theta(n^2)$
- $\Theta(n)$

14.  $n(n-1)/2$  è

- $\Omega(n)$
- $\Omega(n^2)$
- $\Omega(n^2 \log(n))$
- $\Theta(n^2)$
- $\Theta(n)$

15.  $20 \log(n) + 36 n^2 \log(n)$  è

- $\Omega(n)$
- $\Omega(n^2)$
- $\Omega(n^2 \log(n))$
- $\Theta(n^2)$
- $\Theta(n)$

16. Se il limite per  $n$  che va all'infinito di  $f(n)/g(n) = 0$  possiamo dire che

- $f(n)$  è  $O(g(n))$
- $f(n)$  è  $o(g(n))$
- $f(n)$  è  $\Omega(g(n))$
- $g(n)$  è  $\Omega(f(n))$
- $g(n)$  è  $O(f(n))$
- niente

17. Se  $f(n)$  è  $O(g(n))$  allora possiamo dire che

- il limite per  $n$  che va all'infinito di  $f(n)/g(n) = 0$
- il limite per  $n$  che va all'infinito di  $f(n)/g(n) = \text{costante maggiore di } 0$
- niente
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è più infinito
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è finito
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  non esiste

18. Se  $f(n)$  è  $\Theta(g(n))$  allora possiamo dire che

- il limite per  $n$  che va all'infinito di  $f(n)/g(n) = 0$
- il limite per  $n$  che va all'infinito di  $f(n)/g(n) = \text{costante maggiore di } 0$
- niente
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è più infinito
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è finito
- il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  non esiste

19. Se  $f(n) \in o(g(n))$  allora possiamo dire che
- il limite per  $n$  che va all'infinito di  $f(n)/g(n) = 0$
  - il limite per  $n$  che va all'infinito di  $f(n)/g(n) = \text{costante maggiore di } 0$
  - niente
  - il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è più infinito
  - il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  è finito
  - il limite per  $n$  che va all'infinito di  $f(n)/g(n)$  non esiste
20. Insertion-sort nel caso medio ha un costo computazionale
- $O(n)$
  - $O(n^2)$
  - $O(n \log(n))$
  - $o(n^2)$
21. Insertion-sort nel caso medio ha un costo computazionale
- $O(n)$
  - $O(n^2)$
  - $O(n \log(n))$
  - $o(n^2)$
22. Insertion-sort nel caso ottimo ha un costo computazionale
- $O(\log(n))$
  - $O(n^2)$
  - $o(n)$
  - $o(n^2)$
23. Per definire una struttura dati astratta dobbiamo
- definire i dati e le operazioni su di essi
  - descrivere le strutture dati che contengono i dati
  - descrivere l'implementazione delle operazioni sui dati
24. Quali delle seguenti sono strutture dati astratte?
- un insieme di interi con l'operazione "estrai il massimo"
  - un vettore di  $n$  numeri con l'operazione "estrai il massimo"
  - un vettore ordinato di  $n$  numeri con l'operazione "estrai il massimo"
  - un heap con l'operazione "estrai il massimo"
25. Nel progettare una struttura dati "buona" si cerca di
- minimizzare le risorse usate (tempo, spazio di memoria, ecc.) in funzione delle risorse disponibili mediamente negli elaboratori in commercio al momento
  - minimizzare le risorse usate (tempo, spazio di memoria, ecc.) in funzione delle risorse che si prevede saranno disponibili negli elaboratori entro qualche anno
  - massimizzare le risorse usate (tempo, spazio di memoria, ecc.) in funzione delle risorse disponibili mediamente negli elaboratori attuali
  - minimizzare le risorse usate (tempo, spazio di memoria, ecc.) in funzione della dimensione dell'istanza del problema che vogliamo risolvere

26. L'approccio divide et impera ha il seguente costo computazionale:

- sempre polinomiale
- sempre esponenziale
- dipende dal problema
- non si può dire in anticipo

27. Merge-sort ordina un vettore  $A[1,..,n]$

- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  è circa  $n/2$ , e costruendo un nuovo vettore ordinato dato dalla loro unione
- selezionando ad ogni ciclo l'elemento minimo della porzione di vettore non ancora ordinata  $A[j,..,n]$  per metterlo nella posizione  $j$
- selezionando ad ogni ciclo l'elemento massimo della porzione di vettore non ancora ordinata  $A[1,..,i]$  per metterlo nella posizione  $i$
- mantenendo un heap nella prima parte  $A[1,..,i]$  del vettore, scambiando ad ogni ciclo la radice dello heap con l'elemento in posizione  $i$
- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$  ottenendo il vettore ordinato alla fine del processo

28. Heap-sort ordina un vettore  $A[1,..,n]$

- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  è circa  $n/2$ , e costruendo un nuovo vettore ordinato dato dalla loro unione
- selezionando ad ogni ciclo l'elemento minimo della porzione di vettore non ancora ordinata  $A[j,..,n]$  per metterlo nella posizione  $j$
- selezionando ad ogni ciclo l'elemento massimo della porzione di vettore non ancora ordinata  $A[1,..,i]$  per metterlo nella posizione  $i$
- mantenendo un heap nella prima parte  $A[1,..,i]$  del vettore, scambiando ad ogni ciclo la radice dello heap con l'elemento in posizione  $i$
- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  dipende dall'implementazione, ottenendo il vettore ordinato alla fine del processo

29. Quick-sort ordina un vettore  $A[1,..,n]$

- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  è circa  $n/2$ , e costruendo un nuovo vettore ordinato dato dalla loro unione
- selezionando ad ogni ciclo l'elemento minimo della porzione di vettore non ancora ordinata  $A[j,..,n]$  per metterlo nella posizione  $j$
- selezionando ad ogni ciclo l'elemento massimo della porzione di vettore non ancora ordinata  $A[1,..,i]$  per metterlo nella posizione  $i$
- mantenendo un heap nella prima parte  $A[1,..,i]$  del vettore, scambiando ad ogni ciclo la radice dello heap con l'elemento in posizione  $i$
- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  dipende dall'implementazione ottenendo il vettore ordinato alla fine del processo

30. Insertion-sort ordina un vettore  $A[1,..,n]$
- ordinando ricorsivamente i sottovettori  $A[1,q]$ ,  $A[q+1,n]$ , dove  $q$  è circa  $n/2$ , e costruendo un nuovo vettore ordinato dato dalla loro unione
  - selezionando ad ogni ciclo l'elemento minimo della porzione di vettore non ancora ordinata  $A[j,...,n]$  per metterlo nella posizione  $j$
  - selezionando ad ogni ciclo l'elemento massimo della porzione di vettore non ancora ordinata  $A[1,...,i]$  per metterlo nella posizione  $i$
  - mantenendo un heap nella prima parte  $A[1,...,i]$  del vettore, scambiando ad ogni ciclo la radice dello heap con l'elemento in posizione  $i$
  - mantenendo un vettore ordinato  $A[1,...,j-1]$  nel quale ad ogni ciclo viene aggiunto l'elemento  $j$ -esimo del vettore
31. Merge-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- stabile
  - non stabile
32. Heap-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- stabile
  - non stabile
33. Quick-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- stabile
  - non stabile
34. Insertion-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- stabile
  - non stabile
35. Merge-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- in place
  - non in place
  - dipende dall'implementazione
36. Heap-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- in place
  - non in place
  - dipende dall'implementazione
37. Quick sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- in place
  - non in place
  - dipende dall'implementazione
38. Insertion-sort (nella versione dei lucidi presentati a lezione) è un algoritmo
- in place
  - non in place
  - dipende dall'implementazione

39. Merge-sort nel caso pessimo ha costo computazionale

- $O(n \log(n))$
- $O(n^2)$
- $\Theta(n^2)$
- $o(n \log(n))$
- $o(n^3)$

40. Quick-sort nel caso pessimo ha costo computazionale

- $O(n \log(n))$
- $O(n^2)$
- $\Theta(n^2)$
- $o(n \log(n))$
- $o(n^3)$

41. Heap-sort nel caso pessimo ha costo computazionale

- $O(n \log(n))$
- $\Theta(n^2)$
- $\Theta(n^2)$
- $o(n \log(n))$
- $o(n^3)$

42. Il principio di induzione dice che una affermazione è vera

- per ogni  $n >= 0$  se è vera per ogni  $n <= k$  con  $k$  molto grande
- per ogni  $n >= 0$  se è vera per  $0$  e  $\text{affermazione}(n-1)$  implica  $\text{affermazione}(n)$
- per ogni  $n 0$
- per ogni  $n >= 0$  se è vera per  $n <= k$  e  $\text{affermazione}(n)$  implica  $\text{affermazione}(n+1)$  per ogni  $n >$

43. Il problema di ordinare  $n$  numeri (per input generali) ha una complessità computazionale

- $\Theta(n \log(n))$
- $\Theta(n^2)$
- $O(n \log(n))$
- $\Theta(n^3)$
- $\Omega(n \log(n))$

44.  $T(n) = T(n/2) + 1$  ha soluzione

- $T(n) = \Theta(n^2)$
- $T(n) = \Theta(n \log(n))$
- $T(n) = \Theta(n)$
- $T(n) = \Theta(\log(n))$

45.  $T(n) = 2T(n/2) + n$  ha soluzione

- $T(n) = \Theta(n^3/2)$
- $T(n) = \Theta(n \log(n))$
- $T(n) = \Theta(n)$
- $T(n) = \Theta(\log(n))$

46. Ricercare un numero in un vettore ha complessità computazionale

- $\Theta(n \log(n))$
- $\Theta(n^2)$
- $\Theta(n)$
- $O(n \log(n))$
- $\Theta(n)$

47. Ricercare un numero in un vettore ordinato ha complessità computazionale

- $\Theta(\log(n))$
- $\Theta(n^2)$
- $\Theta(n)$
- $O(n \log(n))$
- $\Theta(n)$

48. Dato un vettore di  $n$  elementi nel quale si devono fare  $k$  ricerche, con  $k < \log n$

- conviene fare le ricerche senza ordinare il vettore
- conviene ordinare il vettore prima di fare le ricerche
- conviene ordinare il vettore dopo aver fatto le ricerche

49. Dato un vettore non ordinato di  $n$  elementi nel quale si devono fare  $k$  ricerche

- conviene sempre ordinare il vettore prima di fare la ricerca
- se  $k > n$  conviene fare le ricerche senza ordinare il vettore
- se  $k = n$  conviene ordinare il vettore dopo aver fatto le ricerche
- se  $k > n$  conviene ordinare il vettore prima di fare le ricerche
- non conviene mai ordinare il vettore

50. Che costo computazionale ha ricercare il massimo in una priority queue?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(\log(n))$
- Dipende dalla implementazione

51. Che costo computazionale ha ricercare il massimo in una priority queue implementata con un vettore non ordinato?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(\log(n))$

52. Che costo computazionale ha ricercare il massimo in una priority queue implementata con un vettore ordinato?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(\log(n))$

53. Che costo computazionale ha ricercare il massimo in una priority queue implementata con un heap?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
54. Che costo computazionale ha estrarre il massimo in una priority queue implementata con un vettore non ordinato?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
55. Che costo computazionale ha estrarre il massimo in una priority queue implementata con un vettore ordinato?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
56. Che costo computazionale ha estrarre il massimo in una priority queue implementata con un heap?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
57. Che costo computazionale ha inserire un elemento in una priority queue implementata con un vettore non ordinato?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
58. Che costo computazionale ha inserire un elemento in una priority queue implementata con un vettore ordinato?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
59. Che costo computazionale ha inserire un elemento in una priority queue implementata con un heap?
- $\Theta(n)$   
  $\Theta(1)$   
  $\Theta(\log(n))$
60. Che costo computazionale ha la procedura Build-heap?
- $\theta(n)$   
  $\theta(1)$   
  $\theta(n \log n)$   
 Altro  
  $\theta(n^2)$

61. Che costo computazionale ha la procedura Heapify?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(n \log(n))$
- $\Theta(n^2)$
- Altro

62. Che costo computazionale ha la procedura Partition usata da Quick-sort?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(n \log(n))$
- $\Theta(n^2)$
- Altro

63. Che costo computazionale ha Quick-sort nel caso medio?

- $\Theta(n)$
- $\Theta(1)$
- $\Theta(n \log(n))$
- $\Theta(n^2)$
- Altro

64. E possibile ordinare n numeri in tempo  $O(n \log(n))$ ?

- Dipende dalle proprietà dell'input
- Per input generali no
- sì, sempre
- sì, ma nel caso medio

65. Per algoritmi comparison sort esiste un limite inferiore di complessità (caso pessimo, input generali) pari a

- $\Omega(n)$
- $\Omega(1)$
- $\Omega(n \log(n))$
- $\Omega(n^2)$

66. Quali dei seguenti metodi permettono di risolvere equazioni ricorsive?

- Albero di sostituzione
- Albero di ricorsione
- Master method
- Metodo di intersezione
- Metodo di sostituzione

67. Cosa è una priority queue?

- Una struttura dati astratta
- Una struttura dati concreta
- Un grafo
- Un albero

68. Quali operazioni possono essere eseguite in una priority queue ?

- Estrazione massimo
- Inserimento
- Ordinamento

69. Un albero con  $n$  nodi è definito come

- un grafo aciclico con  $n-1$  archi
- un grafo aciclico connesso con  $n-1$  archi
- un grafo connesso con  $n-1$  archi
- un grafo connesso ciclico

70. La radice di un albero è quel nodo che

- non ha figli
- non ha padre
- ha più figli di tutti gli altri nodi
- ha più fratelli di tutti gli altri nodi

71. Una foglia di un albero è un nodo che

- non ha figli
- non ha padre
- ha più figli di tutti gli altri nodi
- ha più fratelli di tutti gli altri nodi

72. L'altezza  $h(a)$  di un nodo  $a$  è data

- dal numero di figli del nodo  $a$
- dalla distanza tra il nodo  $a$  e la radice
- dalla distanza massima tra il nodo  $a$  ed una foglia
- nessuna delle precedenti

73. Un albero si dice binario quando

- non ha più di due nodi
- ogni nodo non ha più di due figli
- l'altezza dell'albero è sempre multipla di due
- il numero di nodi dell'albero è potenza di due

74. Un heap può essere definito come

- un albero completo
- un albero binario completo
- un albero binario in cui il numero di nodi non supera mai il numero di archi
- nessuna delle precedenti

75. In un heap la relazione tra nodi figli dello stesso padre è

- il figlio sinistro contiene una chiave minore delle chiavi dei figli destri
- il figlio destro contiene una chiave minore delle chiavi dei figli sinistri
- nessuna relazione
- il figlio destro contiene una chiave uguale alla chiave del figlio sinistro

76. Il costo computazionale della procedura Heapify è
- $\Theta(n)$
  - $\Theta(1)$
  - $\Theta(\log(n))$
  - nessuna delle precedenti
77. Il costo computazionale per estrarre il massimo in una priority queue implementata con un heap è
- $O(n)$
  - $O(1)$
  - $O(\log(n))$
  - $\Theta(n)$
78. Il costo computazionale per inserire un elemento in una priority queue implementata con un heap è
- $O(n)$
  - $O(1)$
  - $O(\log(n))$
  - $\Theta(n)$
79. Cosa è un algoritmo comparison sort?
- un algoritmo per ordinare in tempo lineare
  - un algoritmo per ordinare che confronta i numeri tra di loro
  - un algoritmo per ordinare il cui output dipende solo dall'esito di confronti tra numeri
80. Un algoritmo di ordinamento si dice stabile se
- funziona sempre anche quando ci sono malfunzionamenti hardware
  - produce un output in cui se due elementi uguali dell'input erano in un certo ordine vi rimangono
  - funziona senza usare memoria aggiuntiva oltre a quella usata per memorizzare l'input
  - funziona senza usare memoria aggiuntiva oltre a quella usata per memorizzare l'input e l'output
81. Un algoritmo di ordinamento ordina in place se
- funziona sempre anche quando ci sono malfunzionamenti hardware
  - produce un output in cui se due elementi uguali dell'input erano in un certo ordine vi rimangono
  - funziona senza usare memoria aggiuntiva oltre a quella usata per memorizzare l'input
  - funziona senza usare memoria aggiuntiva oltre a quella usata per memorizzare l'input e l'output
82. Quali delle seguenti affermazioni è vera?
- calcolare il massimo in un vettore ha una complessità superiore a calcolare il mediano
  - calcolare il massimo in un vettore ha una complessità inferiore a calcolare il mediano
  - calcolare il massimo in un vettore ha la stessa complessità di calcolare il mediano
83. Quali delle seguenti affermazioni è vera?
- calcolare il massimo in un vettore ha una complessità superiore a calcolare l'i-esimo elemento nell'ordinamento
  - calcolare il massimo in un vettore ha una complessità inferiore a calcolare l'i-esimo elemento nell'ordinamento
  - calcolare il massimo in un vettore ha la stessa complessità di calcolare l'i-esimo elemento nell'ordinamento

84. È possibile calcolare l'i-esimo elemento nell'ordinamento partendo da un vettore non ordinato in tempo lineare
- Si
  - No
  - Dipende
  - Si, ma solo se i valori sono monotoni
85. In una pila gli elementi vengono estratti utilizzando una politica
- LIFO
  - FIFO
  - TIFO
  - SCHIFO
86. In una coda gli elementi vengono estratti utilizzando una politica
- LIFO
  - FIFO
  - TIFO
  - SCHIFO
87. Qual è il problema nell'usare i vettori per implementare code e pile?
- non posso avere strutture dati di dimensioni maggiori della memoria presente nel calcolatore in uso
  - devo indirizzare gli elementi usando gli indici del vettore
  - non posso avere strutture dati di dimensioni maggiori di quelle definite in fase di programmazione
  - le operazioni di ordinamento e selezione sono computazionalmente molto costose
88. In una doubly linked list cancellare un elemento (fornito tramite un puntatore) costa
- $O(n)$
  - $O(n \log(n))$
  - $O(1)$
89. In una singly linked list cancellare un elemento (fornito tramite un puntatore) costa
- $O(n)$
  - $O(n \log(n))$
  - $O(1)$
90. Ricercare un elemento in una singly linked list ordinata costa
- $O(n)$
  - $O(\log(n))$
  - $O(1)$
91. Ricercare un elemento in una doubly linked list ordinata costa
- $O(n)$
  - $O(\log(n))$
  - $O(1)$

92. In una doubly linked list inserire un elemento (fornito tramite un puntatore) costa
- O(n)
  - O(nlog(n))
  - O(1)
93. In una doubly linked list ordinata inserire un elemento (fornito tramite un puntatore) costa
- O(n)
  - O(nlog(n))
  - O(1)
94. In una coda implementata tramite doubly linked list ENQUEUE costa
- O(n)
  - O(nlog(n))
  - O(1)
95. In una coda implementata tramite doubly linked list DEQUEUE costa
- O(n)
  - O(nlog(n))
  - O(1)
96. In una coda implementata tramite doubly linked list con sentinelle DEQUEUE costa
- $O(n)$
  - $O(n \log n)$
  - $O(1)$
97. In una pila implementata tramite doubly linked list PUSH o POP costano
- O(n)
  - O(nlog(n))
  - O(1)
  - hanno costi diversi
98. Come posso rappresentare alberi generali tramite liste?
- ogni nodo ha un puntatore al padre, al figlio sinistro ed alla lista dei fratelli
  - ogni nodo ha un puntatore per ogni figlio ed uno al padre
  - la radice ha un puntatore per ogni foglia ed ogni foglia ha un puntatore alla radice
  - ogni nodo ha un puntatore al padre ed uno al fratello destro
99. Come posso rappresentare alberi binari più semplicemente che con la rappresentazione per gli alberi generali?
- ogni nodo ha un puntatore al padre, al figlio sinistro ed alla lista dei fratelli
  - ogni nodo ha un puntatore per ogni figlio ed uno al padre
  - la radice ha un puntatore per ogni foglia ed ogni foglia ha un puntatore alla radice
  - ogni nodo ha un puntatore al padre ed uno al fratello destro
100. Perché si implementano le liste usando le sentinelle?
- perché così facendo si diminuisce il costo computazionale
  - perché il codice risulta più semplice e pulito
  - perché è necessario farlo

101. In una tabella hash con indirizzamento diretto le operazioni di inserimento, ricerca e cancellazione hanno un costo computazionale

- costante
- lineare
- logaritmico
- quadratico

102. In una tabella hash con funzione hash  $h$ , la chiave  $i$  viene memorizzata in posizione

- $i$
- $h(i)$
- $ih(i)$
- all'inizio della tabella

103. Una buona funzione hash deve

- minimizzare le collisioni
- minimizzare lo spazio di memoria che serve per essere calcolata
- massimizzare le collisioni
- essere veloce da calcolare

104. Quando si genera una collisione in una tabella hash?

- quando la memoria della tabella viene esaurita
- quando si associa la stessa posizione nella tabella a due chiavi distinte
- quando due chiavi distinte hanno molti bit in comune
- quando la funzione hash non è suriettiva

105. Una buona funzione hash può essere iniettiva

- sì
- no
- solo se la chiavi appartengono ad un dominio numerabile

106. Se una funzione hash è biettiva

- ho troppe collisioni
- non ho collisioni, ma la tabella risulta essere di dimensioni troppo grandi
- non riesco a inserire elementi nella tabella in tempo costante

107. Quali delle seguenti proprietà sono gradite per una funzione hash?

- iniettività
- suriettività
- assomigliare a una funzione random
- generare posizioni che dipendono da tutti i bit della chiave
- nessuna delle precedenti

108. Cosa si intende per simple uniform hashing ?

- la tabella si riempie in modo uniforme
- le chiavi vengono generate in modo uniforme
- la probabilità che si inserisca un elemento nella posizione  $i$  della tabella è uguale alla probabilità che si inserisca un elemento nella posizione  $j$ , per ogni  $i,j$
- la funzione hash è costante

109. Usando il metodo di chaining nella gestione dei conflitti

- una tabella con m posizioni può contenere al massimo m elementi
- una tabella con m posizioni può contenere più di m elementi

110. Il load factor in una hash table è

- il rapporto tra dimensione della tabella e numero di elementi contenuti nella tabella
- dimensione della tabella
- numero di elementi massimo che può contenere la tabella
- il rapporto tra il numero di elementi nella tabella e la dimensione della tabella
- nessuno dei precedenti

111. Il costo computazionale medio per la ricerca di un elemento in una tabella hash dove le collisioni vengono gestite con la tecnica di chaining è

- $\Theta(1 + \text{load factor})$
- $\Theta(1 + \text{load factor})$  ma solo nel caso di simple uniform hashing
- $\Theta(\text{load factor} + \text{dimensione della tabella})$
- $\Theta(\text{load factor} + \text{numero di elementi presenti nella tabella})$

112. Il costo computazionale per la ricerca di un elemento in una tabella hash dove le collisioni vengono gestite con la tecnica di chaining nel caso pessimo è

- $\Theta(\text{numero di elementi presenti nella tabella})$
- $\Theta(1 + \text{load factor})$
- $\Theta(\text{load factor} + \text{dimensione della tabella})$
- $\Theta(\text{load factor} + \text{numero di elementi presenti nella tabella})$

113. Il costo computazionale per la ricerca di un elemento in una tabella hash dove le collisioni vengono gestite con la tecnica di chaining nel caso medio con simple uniform hashing e  $n=O(m)$ , cioè il numero di elementi inseriti proporzionale alle dimensioni d

- $O(n)$
- $O(n \log(n))$
- $O(1)$

114. Per avere uniform hashing, nota la probabilità  $Pr(k)$  di ogni chiave, la somma delle probabilità delle chiavi che collidono ( $\sum Pr(k)$  per ogni  $k$  t.c.  $h(k)=j$ ) deve essere

- uguale alla somma delle probabilità delle altre chiavi che collidono: se  $m$  è la dimensione della tabella  $\sum Pr(k) = 1/m$
- uguale alla media delle probabilità: se  $n$  è il numero di chiavi che collidono  $\sum Pr(k) = \sum Pr(k)/n$
- uguale alla probabilità massima delle chiavi che collidono:  $\sum Pr(k) = \max(Pr(k))$
- non è un valore che influisce sull'uniform hashing

115. Nella progettazione di funzioni hash che godano di uniform hashing se non sono note a priori le probabilità delle chiavi  $Pr(k)$

- non è possibile usare una funzione hash per indirizzarle
- si usano delle euristiche facendo dipendere  $h$  da tutti i bit di  $k$  cercando di mantenere indipendenza da pattern particolari
- si può usare solo una funzione  $h(k)=k$

- si usano delle euristiche facendo dipendere  $h$  da eventuali pattern presenti nelle chiavi

116. Data una chiave  $k$  ed una tabella di dimensione  $m$  il Metodo della divisione prevede una funzione hash

- $h(k) = k$
- $h(k) = k \bmod m$
- $h(k) = \text{Parte\_Intera\_Inferiore}[m(kA \bmod m)]$
- $h(k) = \text{Parte\_Intera\_Inferiore}[km]$
- nessuna delle precedenti

117. In una hash table con gestione delle collisioni tramite open addressing

- il load factor non può mai essere maggiore di 1
- il load factor non può mai essere minore di 1
- il load factor può anche essere maggiore di 1

118. In una hash table (uniform hashing) con gestione delle collisioni tramite open addressing la lunghezza media di una probe

- $1/(1 - \text{load factor})$
- $1 + \text{load factor}$
- load factor
- dimensione della tabella + 1

119. Sia  $x$  un nodo di un albero binario di ricerca

- la chiave del figlio sinistro di  $x$  è minore o uguale della chiave di  $x$
- la chiave del figlio destro di  $x$  è maggiore della chiave di  $x$
- la chiave di  $x$  è maggiore o uguale della chiave dei figli di  $x$
- la chiave di  $x$  è minore o uguale della chiave dei figli di  $x$
- nessuna delle precedenti

120. la ricerca di un elemento in un albero binario di ricerca con  $n$  nodi di altezza  $h$  ha un costo computazionale

- $O(n)$
- $O(\log(n))$
- $O(h)$
- $O(\log(h))$
- $O(n\log(h))$

121. l'inserimento di un elemento in un albero binario di ricerca con  $n$  nodi di altezza  $h$  ha un costo computazionale

- $O(n)$
- $O(\log(n))$
- $O(h)$
- $O(\log(h))$
- $O(n\log(h))$

122. la cancellazione di un elemento in un albero binario di ricerca con n nodi di altezza h ha un costo computazionale

- O(n)
- O(log(n))
- O(h)
- O(log(h))
- O(nlog(h))

123. Sia x un nodo di un albero binario di ricerca. Il successore di x è

- il figlio destro di x
- il minimo nel sottoalbero sinistro di x
- il minimo nel sottoalbero destro di x
- il massimo del sottoalbero sinistro di x

124. Il minimo in un albero binario di ricerca

- è una foglia
- ha al più un figlio
- ha almeno un figlio
- ha più di due figli

125. Il successore di un nodo in un albero di ricerca viene usato quando

- si inserisce un nodo più grande del massimo
- si cerca un nodo
- si cancella un nodo senza figli
- si cancella un nodo con due figli

126. L'altezza di un albero binario di ricerca con n nodi

- è al massimo log(n)
- è al massimo n-1
- è al massimo radice quadrata di n-1
- è costante

127. Quali delle seguenti affermazioni relative alla programmazione dinamica sono vere?

- Lo stesso sottoproblema può essere risolto più volte
- Ogni sottoproblema viene risolto una sola volta e il risultato memorizzato in una tabella
- La soluzione di un sottoproblema viene memorizzata per un certo lasso di tempo e poi viene rimossa dalla tabella

128. Quali dei seguenti sono passi fondamentali in programmazione dinamica

- calcolo delle soluzioni per tutti i possibili sottoproblemi
- caratterizzazione della struttura di una soluzione ottima
- definizione ricorsiva del valore di una soluzione ottima
- enumerazione di tutti i possibili sottoproblemi
- costruzione di una soluzione ottima a partire dalle informazioni già calcolate

129. Dire se la seguente affermazione è una condizione necessaria per applicare la programmazione dinamica: una soluzione ottima per il problema contiene al suo interno le soluzioni ottime dei sottoproblemi.

- VERO
- FALSO
- dipende

130. Cosa significa che un problema di ottimizzazione ha sottoproblemi comuni?

- significa che un algoritmo ricorsivo richiede di risolvere più di una volta lo stesso sottoproblema
- significa che nella soluzione ottima ci sono contenute le sottosoluzioni dello stesso sottoproblema più volte
- significa che per risolvere il problema dobbiamo risolvere più di un sottoproblema
- significa che tutti i sottoproblemi sono uguali e quindi è sufficiente risolverne una sola copia

131. Tutti i problemi di ottimizzazione possono essere risolti efficientemente con la programmazione dinamica.

- vero
- falso
- tutti, ma solo quelli che godono della proprietà cumulativa

132. Tutti i problemi di ottimizzazione possono essere risolti efficientemente con l'approccio divide et impera

- VERO
- FALSO
- Tutti, ma solo quelli che godono della proprietà di divisionalità

133. Quali delle seguenti affermazioni meglio descrivono un algoritmo greedy ?

- un algoritmo greedy risolve alcuni sottoproblemi e poi calcola la soluzione finale usando le informazioni calcolate
- un algoritmo greedy compie una sequenza di scelte basandosi sui dati a disposizione e cercando ad ogni passo di costruire la migliore soluzione possibile
- un algoritmo greedy calcola una soluzione e poi cerca di migliorarla fino a trovare quella ottima

134. Gli algoritmi greedy trovano sempre la soluzione ottima.

- vero
- falso
- dipende dall'input

135. Il seguente algoritmo per il problema del commesso viaggiatore è un algoritmo greedy? Parto dalla città numero 1 e procedo visitando la città più vicina non ancora visitata. Quando tutte le città sono state visitate torno alla città numero 1.

- si
- no
- dipende
- solo se le distanze sono euclidee
- solo se le distanze sono non-negative

136. Esiste un algoritmo greedy per risolvere il commesso viaggiatore?

- si
- no
- non si sa

137. E vero che, come nel caso della programmazione dinamica, anche per applicare un algoritmo greedy occorre che la soluzione ottima contenga le soluzioni ottime dei sottoproblemi?

- si
- no
- dipende

138. Quali delle seguenti affermazioni sono vere?

- e sempre possibile risolvere un problema di ottimizzazione con algoritmi greedy o con programmazione dinamica
- non e sempre possibile risolvere un problema di ottimizzazione con algoritmi greedy o con programmazione dinamica
- ci sono problemi che si possono risolvere con programmazione dinamica ma non si possono risolvere con algoritmi greedy
- i problemi che si possono risolvere con programmazione dinamica si possono risolvere anche con algoritmi greedy

139. I problemi di ottimizzazione che soddisfano la proprietà della sottostruttura ottima possono essere risolti con algoritmi greedy

- vero
- falso
- dipende

140. I problemi di ottimizzazione che soddisfano la proprietà della sottostruttura ottima possono essere risolti con la programmazione dinamica

- vero
- falso
- dipende

141. I problemi di ottimizzazione che soddisfano la proprietà della sottostruttura ottima possono essere risolti sia con algoritmi greedy che con la programmazione dinamica

- vero
- falso
- dipende

142. Un algoritmo greedy ha il seguente costo computazionale:

- sempre polinomiale
- sempre esponenziale
- dipende dal problema
- nessuno dei precedenti

143. L'approccio della programmazione dinamica ha il seguente costo computazionale:

- sempre polinomiale
- sempre esponenziale
- dipende dal problema
- nessuno dei precedenti

144. I numeri di fibonacci sono definiti come

- $F_0=1, F_1=1, F_k=F_{k-1} \cdot F_{k-2}$
- $F_0=0, F_1=0, F_k=F_{k-1} + F_{k-2}$
- $F_0=0, F_1=1, F_k=F_{k-1} - F_{k-2}$
- $F_0=0, F_1=1, F_k=F_{k-1} + F_{k-2}$
- $F_0=0, F_1=1, F_k=F_{k-1} + F_{k-2}$

145. Per i numeri di fibonacci vale che

- $F_{k+2} = F_k + F_{k+1} \quad \forall k \geq 0$
- $F_k = F_{k-2} + F_{k-1} \quad \forall k \geq 2$
- $F_{k+2} = 1 + \sum_{i=0, \dots, k} F_i \quad \forall k \geq 0$
- $F_{k+1} = F_k + F_{k-1} \quad \forall k \geq 1$
- $F_{k+2} = F_k * F_{k+1} \quad \forall k \geq 0$

146. Un insieme disgiunto è

- Un insieme in cui sono stati esplicitamente definiti dei sottinsiemi
- Un insieme partizionato in sottinsiemi
- Un insieme i cui sottinsiemi sono degli up-tree
- Un insieme che non gode della proprietà di congiunzione
- Un insieme che è stato staccato da un altro insieme

147. Le operazioni principali per gli insiemi disgiunti sono

- Extract-Min(x), Make-Set(x), Union(x,y)
- Make-Set(x), Union(x,y), Delete(x,T)
- Make-Root(x,T), Union(x,y), Delete(x,T)
- Make-Set(x), Union(x,y), Find-Set(x)
- Disjunct(x,T), Union(x,y), UpTree(x,T)

148. La funzione Make-Set(x) per insiemi disgiunti

- inizializza un nuovo insieme contenente il solo elemento x
- crea un nuovo insieme vuoto puntato da x
- prende l'elemento x da un insieme e crea un insieme disgiunto contenente x
- crea l'insieme x

149. La funzione Union(x,y) per gli insiemi disgiunti

- unisce gli insiemi x e y in un unico insieme  $x \cup y$
- crea un nuovo insieme contenente x e y
- verifica se x e y sono insieme disgiunti, se non lo sono li unisce
- unisce gli elementi degli insiemi che contengono x e y, S e T rispettivamente, nell'unico insieme  $S \cup T$
- unisce l'elemento x agli elementi dell'insieme y

150. La funzione Find-Set(x) per insiemi disgiunti

- trova l'insieme x fra tutti gli insiemi disgiunti
- trova l'insieme x e verifica che sia disgiunto dagli altri
- trova l'insieme a cui appartiene l'elemento x
- trova l'insieme di cui x è la radice
- trova x

151. Gli up-tree possono rappresentare insiemi disgiunti

- collegando i diversi insiemi disgiunti come rami diversi di un up-tree
- associando un up-tree ad ogni insieme disgiunto
- collegando i diversi insiemi disgiunti come foglie diverse di un up-tree
- Facendo risalire verso l'alto nell'up-tree i diversi insiemi disgiunti
- Associando una diversa radice dello stesso up-tree ad ogni insieme disgiunto

152. In un up-tree

- la radice è l'elemento più basso e la ricerca avviene verso l'alto
- la radice contiene il rappresentante di un insieme, che è padre di se stesso
- le foglie puntano direttamente alla radice
- ogni elemento contiene un campo chiave e un puntatore al padre
- ogni elemento punta solo al padre

153. L'unione di due up-tree

- si realizza facendo puntare dalla radice dell'albero che ha più nodi la radice dell'albero che ha meno nodi
- si realizza inserendo per chiave crescente gli elementi dell'albero con meno nodi fra quelli dell'albero con più nodi
- si realizza facendo puntare dalla radice dell'albero che ha meno nodi la radice dell'albero che ha più nodi
- si realizza inserendo per chiave crescente gli elementi dell'albero con più nodi fra quelli dell'albero con meno nodi
- si realizza collegando l'albero con meno nodi ad una foglia dell'albero con più nodi

154. La procedura di compressione di cammini per up-tree

- crea un nuovo up-tree contenente solo il cammino passato come parametro
- serve nel corso della find-set
- collega gli elementi del cammino passato come parametro riducendone così la lunghezza
- invoca l'algoritmo di Dijkstra per identificare l'up-tree di cammini di lunghezza minima
- fa puntare direttamente alla radice ogni nodo del cammino d'accesso al nodo dato

155. Il rango (rank) associato ad ogni nodo  $x$  di un up-tree rappresenta

- il numero di figli di  $x$
- il limite superiore all'altezza di  $x$
- il limite superiore al numero di archi del cammino più lungo fra  $x$  e una foglia discendente
- il numero complessivo di discendenti di  $x$
- il numero di antenati di  $x$

156. Le procedure che modificano direttamente il rango di un nodo di un up-tree sono

- Make-set, link e union
- Find-set e Link
- Make-Set e Union
- Make-Set, Link, Union e Find-set
- Make-set e Link

157. La funzione definita come  $F(0)=1$  e  $F(i)=2F(i-1)$

- è una funzione  $\Theta(2^n)$
- è una funzione che cresce molto lentamente
- è tale per cui già  $F(5)$  è un valore che eccede tutti i numeri incontrati nella pratica normale
- è una funzione  $O(2^n)$
- è una funzione che non serve a niente

158. Se  $i=\log^*(n)$

- $i$  è tale per cui  $F(i) = n$
- $i$  è il più piccolo intero tale che  $F(i) \geq n$
- $i$  è il più piccolo intero tale che  $\log \log \dots \log(n) \geq 1$ , log ripetuto  $i$  volte
- $i$  è il più piccolo intero tale che  $\log \log \dots \log(n) \leq 1$ , log ripetuto  $i$  volte
- $i$  è il più piccolo intero tale che  $\log \log \dots \log(i) \leq 1$ , log ripetuto  $n$  volte

159. La funzione  $\log^*$

- è una funzione che cresce molto molto lentamente
- è una funzione polinomiale
- è una funzione tale per cui  $\log^*(n) \leq 5$  per ogni numero  $n$  incontrato nella pratica normale
- è una funzione tale per cui  $\log^*(5) \leq n$  per ogni numero  $n$  incontrato nella pratica normale
- è una funzione  $\Theta(n/2)$

160. La funzione di Ackerman  $A(i,j)$  è definita come

- $A(1,j)=1$  per  $j \geq 1$ ,  $A(i,1)=A(i-1,2)$  per  $i \geq 1$ ,  $A(i,j)=A(i-1, A(i,j-1))$  per  $i,j > 1$
- $A(1,j)=j$  per  $j \geq 1$ ,  $A(i,1)=i$  per  $i > 1$ ,  $A(i,j)=A(i-1, j)$  per  $i,j > 1$
- $A(1,j)=j$  per  $j \geq 1$ ,  $A(i,1)=A(i-1,j)$  per  $i > 1$ ,  $A(i,j)=A(i-1,j-1)$  per  $i,j > 1$
- $A(1,j)=2j$  per  $j \geq 1$ ,  $A(i,1)=A(i-1,2)$  per  $i > 1$ ,  $A(i,j)=A(i-1, A(i,j-1))$  per  $i,j > 1$
- $A(1,j)=2j$  per  $j \geq 1$ ,  $A(i,1)=2i$  per  $i > 1$ ,  $A(i,j)=A(i-1*A(i,j-1))$  per  $i,j > 1$

161. L'inversa della funzione di Ackerman  $\alpha(m,n)$ , per  $m \geq n$ , è definita come

- $\alpha(m,n)=1/A(m,n)$
- $\alpha(m,n)=i$  se  $i$  è il più piccolo intero tale per cui  $A(i,\lfloor m/n \rfloor) > \log(n)$
- $\alpha(m,n)=i$  se  $i$  è il più piccolo intero tale per cui  $A(i,\lfloor m/n \rfloor) > n$
- $\alpha(m,n)=i$  se  $i$  è il più piccolo intero tale per cui  $A(m,n) > n$
- $\alpha(m,n)=i$  se  $i$  è il più piccolo intero tale per cui  $A(\lfloor m/n \rfloor) < \log(n)$

162. Per un insieme di up-tree vale che

- per tutte le radici  $x$  di alberi,  $\text{size}[x] \geq 2^{\text{rank}[x]}$
- per tutte le radici  $x$  di alberi,  $\text{size}[x] \geq \text{rank}[x]$
- per tutte le radici  $x$  di alberi,  $\text{size}[x] \leq \text{rank}[x]$
- per tutte le radici  $x$  di alberi,  $\text{size}[x] = 2^{\text{rank}[x]}$
- per tutte le radici  $x$  di alberi,  $\text{size}[x] = \text{rank}[x]$

163. Dato una foresta di up-tree, una sequenza di  $m$  operazioni make-set, union e find-set, con  $n$  operazioni make-set, puo' essere eseguita in tempo

- $O(A(m,n))$
- $O(m \log^* n)$
- $O(\alpha(m,n))$
- $O(2n)$
- $O(A(m,\log^* n))$

164. Un grafo  $G$  è

- una rete di connessione di nodi
- un insieme di archi e vertici
- un insieme di nodi collegati
- una rappresentazione cartesiana di una funzione
- una coppia di insiemi  $V$  e  $E$

165. In un grafo  $G=(V,E)$ , un arco  $a \in E$  è

- una coppia  $\{u,v\}$  di vertici,  $u,v \in V$
- un collegamento fra due vertici  $u$  e  $v$ ,  $u,v \in V$
- una connessione fra una coppia  $\{u,v\}$  di vertici,  $u,v \in V$
- una linea che connette due nodi  $u$  e  $v$ ,  $u,v \in V$

166. Un grafo  $G=(V,E)$  è diretto (orientato) se

- gli archi sono rappresentati da delle frecce e non delle linee
- l'insieme  $V$  è formato da vertici e non da nodi
- gli archi sono coppie ordinate di vertici  $(u,v)$
- gli archi vanno sempre dal nodo di indice minore a quello di indice maggiore
- gli archi puntano verso la radice del grafo

167. Il grado di un vertice  $v$  di un grafo è

- il numero di archi che è necessario percorrere per andare dalla radice a  $v$
- il numero di figli di  $v$
- il numero di vertici connessi da un arco con  $v$
- il numero di vertici adiacenti a  $v$
- il numero di nodi di  $G$

168. Un cammino in un grafo  $G=(V,E)$  è

- un insieme di archi di  $E$
- una sequenza  $v_1, v_2, \dots, v_k$  di vertici di  $G$  tale che ogni coppia di vertici consecutivi  $v_i, v_{i+1}$  sia adiacente
- una sequenza  $v_1, v_2, \dots, v_k$  di vertici di  $G$
- una sequenza  $v_1, v_2, \dots, v_k$  di vertici di  $G$  tale che  $v_1 = v_k$
- un percorso fra vertici di  $G$

169. La somma dei gradi di tutti i vertici di un grafo  $G=(V,E)$  è

- uguale al doppio del numero dei vertici in  $V$
- uguale al numero degli archi in  $E$
- uguale alla somma degli archi in  $E$
- compresa fra  $|E|$  e  $2|E|$
- uguale al doppio del numero degli archi in  $E$

170. In un grafo  $G=(V,E)$  un cammino elementare è

- un cammino in cui non ci sono vertici ripetuti
- un cammino costituito da un unico arco
- un cammino costituito da un unico nodo
- un cammino che parte e ritorna allo stesso nodo
- un cammino strutturalmente molto semplice

171. In un grafo  $G=(V,E)$  un ciclo è

- un qualsiasi cammino che parte e rientra allo stesso nodo
- un cammino costituito da due percorsi chiusi
- un cammino che percorre tutti i nodi di  $G$
- un cammino elementare, tranne che per il primo vertice che coincide con l'ultimo
- l'unico cammino che partendo dalla radice ne rientra

172. Un grafo  $G=(V,E)$  è connesso se

- qualsiasi coppia di vertici in  $V$  è unita da al più un cammino
- qualsiasi coppia di vertici adiacenti ha un arco che li connette
- qualsiasi coppia di vertici in  $V$  è unita da almeno un cammino
- qualsiasi coppia di vertici in  $V$  è unita da esattamente un cammino
- non esistono sottografi disconnessi di  $G$

173. Un sottografo  $G^*$  di un grafo  $G=(V,E)$  è

- un sottinsieme dei vertici e degli archi di  $G$
- un grafo che ha tutti i vertici in  $V$  ma solo un sottinsieme degli archi in  $E$
- un grafo che ha tutti gli archi in  $E$  ma solo un sottinsieme dei vertici in  $V$
- un sottinsieme connesso del grafo  $G$
- un grafo contenente solo i vertici più bassi del grafo  $G$

174. Una componente连通的 G\*=(V\*,E\*) di un grafo G=(V,E) è

- un sottografo connesso di  $G$  tale per cui non è possibile aggiungere vertici in  $G \setminus G^*$  senza perdere la connessione
- un cammino connesso in  $G$
- un sottografo connesso massimale di  $G$
- un qualsiasi sottografo connesso di  $G$
- un sottografo connesso di  $G$  tale per cui l'aggiunta di un vertice in  $G \setminus G^*$  implica l'aggiunta di un arco in  $E \setminus E^*$

175. Un albero è

- un grafo in cui è possibile identificare un nodo radice e dei nodi foglia
- un sottografo connesso massimale di un grafo  $G=(V,E)$
- un sottografo connesso di un grafo  $G=(V,E)$  in cui è possibile identificare un nodo radice e dei nodi foglia
- un insieme di cammini di un grafo  $G=(V,E)$
- un grafo connesso senza cicli

176. Una foresta è

- una collezione di alberi
- un albero disconnesso
- un insieme di alberi disconnessi a coppie
- un insieme di alberi con la radice comune
- un insieme di alberi orientati

177. La lista di adiacenza di un vertice v di un grafo  $G=(V,E)$  è

- una lista di tutti gli archi adiacenti a v
- una lista di tutti i vertici di V per cui esiste almeno un cammino che li connette a v
- una lista di tutti i vertici di V per cui esiste al più un cammino che li connette a v
- una lista di tutti i vertici adiacenti a v
- una lista di tutti i vertici di V per cui esiste esattamente un cammino che li connette a v

178. Lo spazio necessario per rappresentare tutte le liste di adiacenza di vertici di un grafo  $G=(V,E)$  con

$$|V|=n \text{ e } |E|=m \text{ è}$$

- $\Theta(n)$
- $\Theta(m+n)$
- $\Theta(n \log m)$
- non polinomiale
- $O(n^m)$

179. Una matrice di adiacenza per un grafo  $G=(V,E)$  è

- una matrice delle liste di adiacenza di tutti i vertici in  $V$
- una matrice che elenca, per ogni vertice  $v$  in  $V$ , i vertici adiacenti a  $v$
- una matrice che specifica le adiacenze del grafo  $G$
- una matrice  $M$  di variabili intere, in cui  $M[i,j]$  è pari al  $j$ -esimo vertice adiacente al nodo  $i$
- una matrice  $M$  di variabili booleane con una cella per ogni coppia di vertici,  $M[i,j]=1$  sse l'arco  $\{i,j\}$  è nel grafo

180. Lo spazio necessario per contenere la matrice di adiacenza di un grafo  $G$  con  $n$  nodi è

- $O(n^2)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $O(n \log n)$
- $O(n)$

181. Una ricerca per ampiezza (BFS) su un grafo  $G$

- percorre l'intero grafo  $G$  e ne definisce un albero di copertura
- percorre una componente连通的 G e ne definisce un grafo di copertura
- percorre una componente连通的 G e ne definisce un albero di copertura
- percorre l'intero grafo  $G$  e ne definisce un grafo di copertura
- percorre l'intero grafo  $G$  e ne definisce un cammino di copertura

182. Una ricerca per ampiezza (BFS) su un grafo  $G$

- dato un vertice sorgente  $s \in V$  calcola la distanza da  $s$  ad ogni vertice raggiungibile di  $G$
- data la radice  $r$  di  $G$  calcola la distanza da  $r$  ad ogni vertice raggiungibile di  $G$
- data la radice  $r$  di  $G$  calcola la distanza da  $r$  ad ogni vertice di  $G$
- calcola la distanza fra ogni coppia di nodi in  $V$
- dati due nodi  $u,v \in V$  calcola la distanza fra  $u$  e  $v$

183. Durante una ricerca BFS lo stato assumibile da ciascun vertice puo' essere

- non espanso, espanso ma non scoperto, scoperto
- espanso, coperto, scoperto
- coperto, scoperto, completo
- interno, esterno, completo
- non scoperto, scoperto ma non espanso, espanso

184. Durante una ricerca BFS

- alla sorgente  $s$  viene assegnata distanza 1, ai nodi adiacenti ad  $s$  distanza 2, a quelli non scoperti adiacenti a questi ultimi 3, ecc.
- alla sorgente  $s$  viene assegnata distanza 0, ai nodi adiacenti ad  $s$  distanza 1, a quelli non scoperti adiacenti a questi ultimi 2, ecc.
- alla sorgente  $s$  viene assegnata distanza 0, ai nodi connessi ad  $s$  distanza 1, a quelli non scoperti connessi a questi ultimi 2, ecc.
- alla sorgente  $s$  viene assegnata distanza 0, ai nodi adiacenti ad  $s$  distanza 1, a quelli adiacenti a questi ultimi 2, ecc.

185. Durante una ricerca BFS l'etichetta di ogni vertice v corrisponde a

- il minimo numero di archi che è necessario percorrere per andare da s a v
- l'identificativo della componente连通的 di v
- il numero di archi nel cammino da s a v
- la lunghezza del cammino più breve da s a v
- il rango di v

186. L'algoritmo BFS

- inizializza la sorgente s e poi espande ricorsivamente l'albero
- inizializza la sorgente s e poi espande ricorsivamente il grafo
- inizializza tutti i vertici e poi procede ricorsivamente
- inizializza l'albero di copertura e poi lo costruisce in modo connesso
- inizializza lo stato dei vertici, inizializza s e poi espande iterativamente l'albero di copertura

187. Il tempo di CPU dell'algoritmo BFS è

- $O(V+E)$
- $O(V^*E)$
- $\Theta(V \log E)$
- $\Theta(V^*E)$
- $O(V \log E)$

188. Il tempo di CPU dell'algoritmo BFS è

- quadratico (rappresentando il grafo con liste di adiacenza)
- lineare (rappresentando il grafo con liste di adiacenza)
- quadratico (rappresentando il grafo con matrici di adiacenza)
- lineare (rappresentando il grafo con matrici di adiacenza)
- non polinomiale

189. Il sottografo  $G'=(V',\dot{E})$  dei predecessori costruito dalla procedura BFS applicata ad un grafo G a partire da una sorgente s

- è un grafo contenente tutti i vertici raggiungibili da s tale per cui per ogni  $v \in V'$  il cammino da s a v è il cammino minimo da s a v in G
- è un grafo contenente tutti i nodi esplorati da BFS durante la ricerca e tale per cui  $\{u,v\} \in \dot{E}$  sse  $u,v \in V'$
- è un grafo contenente tutti i nodi esplorati da BFS durante la ricerca e tale per cui  $\{u,v\} \in \dot{E}$  sse esiste un cammino da s a v e da s a u in G
- è un albero contenente tutti i vertici raggiungibili da s tale per cui per ogni  $v \in V'$  il cammino da s a v è il cammino minimo da s a v in G
- è un albero contenente tutti gli antenati di ogni vertice v, così come identificati dalla procedura BFS quando applicata a G

190. La ricerca in profondità (DFS) applicata ad un grafo G

- percorre una componente连通的 di G senza mai riconsiderare più volte uno stesso nodo
- percorre una componente连通的 di G effettuando backtrack su nodi già esplorati
- percorre una componente连通的 di G effettuando backtrack sul nodo sorgente s
- percorre l'intero grafo G effettuando backtrack di reinizializzazione della ricerca

- percorre l'intero grafo G in ordine di valori ascendenti delle chiavi associate ai nodi

191. La procedura DFS

- Ha una fase di inizializzazione di tutti i vertici e una fase di esplorazione ricorsiva
- Ha una fase di inizializzazione della sorgente e una di esplorazione iterativa dell'intero grafo G
- ha una fase di inizializzazione di tutti i vertici, una di inizializzazione della sorgente e una di backtrack
- Ha una fase di inizializzazione di tutti i vertici e una di backtrack
- Ha una fase di esplorazione ricorsiva e una di backtrack

192. La procedura DFS( $G, s$ ) si appoggia alla procedura DFS-Visit( $u$ ) che

- ad ogni invocazione inizializza un nuovo albero con radice in  $u$
- esplora iterativamente il sottoalbero radicato in  $u$
- effettua il backtracking nella ricerca DFS su  $G$
- identifica il cammino minimo dalla sorgente  $s$  al nodo  $u$
- visita con procedura DFS il nodo  $u$

193. Al termine della procedura DFS applicata ad un grafo  $G=(V,E)$  ogni nodo  $v \in V$  ha associato

- niente
- una etichetta: la lunghezza del cammino minimo da  $s$  a  $v$
- una etichetta: il tempo di visita di  $v$
- una etichetta: il tempo di fine
- due etichette: tempo di visita e tempo di fine

194. Il tempo di CPU della procedura DFS è

- $\Theta(V \log E)$
- $\Theta(V+E)$
- $O(V \log E)$
- $\Theta(V^*E)$
- $\Theta(E \log V)$

195. Il sottografo dei predecessori  $G'=(V', E')$  costruito dalla procedura DFS applicata a un grafo  $G$

- costituisce l'albero dei cammini minimi dalla sorgente  $s$  ad ogni nodo  $v$  di  $V$
- costituisce l'albero dei cammini minimi dalla sorgente  $s$  ad ogni nodo della componente连通的 di  $G$  in cui si trova  $s$
- costituisce un cammino minimo dalla sorgente  $s$  ad ogni nodo  $v$  di  $V$
- forma una foresta di sottoalberi DF
- forma un albero DF

196. Durante la procedura DFS ogni vertice  $v$  è

- coperto prima di  $f[v]$ , scoperto fra  $f[v]$  e  $d[v]$ , esploso dopo  $d[v]$
- inesplorato prima di  $f[v]$  e esplorato dopo
- bianco prima di  $f[v]$  e grigio dopo
- bianco prima di  $f[v]$ , grigio dopo e nero alla fine
- bianco prima di  $d[v]$ , grigio fra  $d[v]$  e  $f[v]$ , nero dopo  $f[v]$

197. Durante la procedura DFS i vertici grigi

- formano un albero binario, implementabile come una heap binomiale
- formano una catena lineare, implementabile come uno stack
- formano un ciclo, implementabile come uno stack
- formano un sottoalbero dei predecessori, implementabile come una heap di Fibonacci
- formano un sottografo generico di G, implementabile con una matrice di adiacenza

198. Il teorema delle parentesi suggerisce che

- la storia di inizio e fine visita dei veri nodi può essere rappresentata da una espressione ben formata
- gli intervalli di inizio e fine visita dei diversi nodi sono disgiunti
- l'intervallo di visita di un nodo u è contenuto propriamente in quello di un altro nodo v
- l'intervallo di un nodo u discendente di v non interseca l'intervallo di v
- l'intervallo della sorgente è il più stretto fra tutti gli intervalli associati ai noti

199. Il teorema del cammino bianco asserisce che in una foresta DFS di un grafo  $G=(V,E)$  un vertice v è discendente di un vertice u se e

- al tempo  $f[u]$  il vertice v è raggiungibile da u con un cammino di soli archi bianchi
- al tempo  $d[u]$  il vertice v è raggiungibile da u con un cammino di soli archi bianchi
- al tempo  $f[v]$  il vertice v è raggiungibile da u con un cammino di soli archi bianchi
- al tempo  $d[v]$  il vertice v è raggiungibile da u con un cammino di soli archi bianchi
- al tempo  $f[v]$  esiste un cammino di archi bianchi che connette u a v

200. a seguito dell'applicazione di una DFS ad un grafo G gli archi di G possono essere classificati come archi:

- bianchi, neri e grigi
- red e black
- dell'albero, all'indietro, in avanti, di attraversamento
- all'indietro, in avanti e diagonali
- dell'albero, in avanti, di backtrack

201. In una DFS di un grafo non orientato G, ogni arco di G

- è un arco dell'albero oppure un arco di backtrack
- può essere un arco dell'albero, all'indietro, in avanti o di attraversamento
- è un arco dell'albero oppure un arco all'indietro
- può essere un arco bianco, grigio o nero
- è un arco non orientato

202. Un DAG è un grafo

- diretto che non contiene cicli diretti
- diretto e连通的
- diretto e orientato
- diretto che non contiene cicli nella versione non orientata del grafo
- diretto che non contiene cammini minimi

203. Un DAG può essere utilizzato per:

- rappresentare cammini minimi su reti stradali
- rappresentare precedenze fra eventi
- memorizzare alberi DFS
- identificare componenti connesse
- memorizzare pagine su disco rigido

204. Un ordinamento topologico di un DAG permette di

- indurre un ordinamento totale dall'ordinamento parziale rappresentato nel DAG
- specificare le relazioni topologiche esistenti fra i nodi del DAG
- identificare le sottostrutture topologicamente ben definite presenti nell'ordinamento rappresentato dal DAG
- sequenziare le attività rappresentate dal DAG
- identificare un ordinamento parziale fra i nodi del DAG

205. Un ordinamento topologico di un DAG è un ordinamento

- polinomiale dei vertici, tale che per ogni coppia di vertici  $u$  e  $v$  del DAG viene identificato un verso per l'arco  $(u,v)$
- ricorsivo, che ordina lessicograficamente i vertici del DAG
- non polinomiale, che costruisce un DAG a partire da un grafo non diretto e connesso
- lineare dei vertici, tale che per ogni arco  $(u,v)$  del DAG,  $u$  appare prima di  $v$  nell'ordinamento
- logaritmico, che identifica il vertice  $u$  topologicamente più vicino ad un vertice  $v$  dato

206. La procedura Topological-Sort( $G$ )

- è una procedura ricorsiva di ordinamento topologico del grafo  $G$
- identifica un ordinamento topologico di un DAG  $G$  calcolando i tempi di fine visita  $f[v]$  per ogni vertice  $v$  del grafo
- calcola i tempi di inizio e fine visita di ogni nodo del DAG  $G$
- introduce gli archi transitivi in  $G$  che permettono di rappresentare un ordinamento totale
- costruisce una lista concatenata di vertici corrispondente all'ordinamento topologico

207. Un grafo diretto  $G$  è aciclico

- se può essere rappresentato con un DAG
- se una ricerca BFS su  $G$  non produce archi di attraversamento
- se una ricerca DFS su  $G$  non produce archi all'indietro
- se una ricerca DFS su  $G$  termina in tempo polinomiale
- se una ricerca BFS su  $G$  termina in tempo polinomiale

208. In un DAG che rappresenta le relazioni di precedenza fra attività associate ai nodi, l'esistenza di un arco  $(u,v)$  significa che

- l'attività  $u$  può iniziare solo quando  $v$  è completata
- esiste una relazione di ordinamento topologico fra  $u$  e  $v$
- l'attività  $v$  può iniziare solo quando  $u$  è completata
- l'attività  $v$  non può iniziare prima dell'attività  $u$
- le attività  $u$  e  $v$  devono iniziare insieme

209. Il tempo di CPU dell'ordinamento topologico di un DAG  $G=(V,E)$  è

- uguale a quello di DFS( $G$ )
- $O(V)$
- $O(V \log E)$
- $O(E \log V)$
- $O(V+E)$

210. Per la dimostrazione di correttezza dell'ordinamento topologico di un DAG  $G=(V,E)$  si dimostra che

- il tempo di CPU di Topological-Order( $G$ ) è  $O(V+E)$
- se  $(u,v) \in E$  allora  $f[u] < f[v]$
- l'ordinamento topologico è transitivo rispetto all'ordinamento parziale indotto dal DAG
- se  $(u,v) \in E$  allora  $f[u] > f[v]$
- se  $u \in V$  e  $v \in V$  allora  $(u,v) \in E$

211. Una componente fortemente connessa di un grafo orientato  $G=(V,E)$  è

- un sottinsieme di vertici di  $V$  tale che esiste un cammino elementare che li congiunge tutti
- un insieme massimale di vertici  $U \subseteq V$  tale che esiste un cammino elementare che li congiunge tutti
- un insieme massimale di vertici  $U \subseteq V$  tale che  $U$  è un insieme connesso
- un sottinsieme  $U$  di vertici di  $V$  tale che se  $u \in U$  e  $v \in U$  allora ciascuno dei due vertici è raggiungibile dall'altro
- un insieme massimale di vertici  $U \subseteq V$  tale che se  $u \in U$  e  $v \in U$  allora ciascuno dei due vertici è raggiungibile dall'altro

212. Un grafo  $GT=(V,ET)$  è il trasposto di un grafo  $G=(V,E)$  se

- $V=V$  e  $ET = E$
- $ET=\{(u,v):(v,u)\in E\}$
- gli archi in  $ET$  sono i trasposti degli archi in  $E$
- gli archi in  $ET$  sono gli stessi di quelli in  $E$  ma con il senso di percorrenza rovesciato
- i vertici in  $V$  sono estremi sia degli archi in  $E$  che di quelli in  $ET$

213. Un grafo  $G$  e il suo trasposto  $GT$

- hanno le stesse componenti fortemente connesse
- hanno gli stessi archi
- hanno gli stessi cammini elementari
- hanno gli stessi alberi di copertura
- hanno gli stessi DAG

214. L'algoritmo Strongly-Connected-Components( $G$ ), con  $G=(V,E)$

- trova in tempo  $O(V \log E)$  le componenti fortemente connesse di  $G$
- trova in tempo  $O(V+E)$  le componenti fortemente connesse di  $G$
- trova in tempo  $O(E \log V)$  le componenti fortemente connesse di  $G$
- trova in tempo  $O(E \log V)$  le componenti connesse di  $G$
- trova in tempo  $O(V \log E)$  le componenti fortemente connesse del trasposto di  $G$

215. Se due vertici  $u$  e  $v$  sono in una stessa componente fortemente connessa allora

- nessun cammino esce da questa CFC
- nessun cammino fra loro esce da questa CFC
- nessuna altra componente connessa contiene  $u$  e  $v$
- esiste una CFC che contiene sia  $u$  che  $v$
- $u$  appartiene al grafo  $G$  e  $v$  al trasposto di  $G$

216. In una DFS, i vertici di una stessa componente fortemente connessa

- sono tutti foglie dell'albero DFS
- sono tutti collegati fra loro da archi bianchi
- sono posti tutti nello stesso albero DFS
- sono posti in ordine inverso nell'albero DFS
- sono tutti i vertici di  $G$

217. Un avo  $\phi(u)$  di un vertice  $u$  è il vertice  $w$

- più lontano da  $u$
- visitato per primo nella DFS che espande  $w$
- che massimizza  $f[w]$
- che ha il maggior numero di discendenti, fra cui  $w$
- che ha come figlio il padre di  $u$

218. In un grafo orientato  $G=(V,E)$  l'avo  $\phi(u)$  di un qualunque  $u$  in  $V$  in una qualunque visita in profondità di  $G$  è

- un antenato di  $u$
- il padre di  $u$
- il padre del padre di  $u$
- la radice dell'albero DFS (o BFS nel caso di ricerca in ampiezza)
- nessuno dei casi precedenti

219. In ogni visita in profondità di un grafo orientato  $G=(V,E)$ , per ogni vertice  $u$  in  $V$  i vertici  $u$  e  $\phi(u)$

- sono visitati in tempi successivi
- sono vertici espansi
- sono vertici esplosi
- appartengono alla stessa DFS
- appartengono alla stessa CFC

220. In un grafo orientato  $G=(V,E)$  due vertici  $u$  e  $v$  in  $V$  appartengono alla stessa CFC sse

- sono entrambi vertici bianchi
- sono espansi a turno
- hanno lo stesso padre
- hanno lo stesso avo in una visita in profondità di  $G$
- nessuno dei precedenti

221. La correttezza della procedura strongly-connected-components( $G$ ) viene dimostrata

- per ricorsione
- per assurdo
- per induzione
- per deduzione
- per caso
- per infrazione

222. Un albero di copertura minima di un grafo  $G=(V,E)$

- un albero  $T$  che collega tutti i vertici del grafo  $G$
- un albero  $T$  che connette tutti i vertici in  $V$  tale che la somma dei pesi associati agli archi di  $T$  sia minima
- un albero  $T$  che minimizza la somma dei pesi associati agli archi di  $T$
- un albero  $T$  che copre il minor numero possibile di vertici di  $G$
- un albero  $T=(V,\bar{E})$ ,  $\bar{E} \subseteq E$ , tale che la somma dei pesi associati agli archi di  $T$  sia minima

223. L'algoritmo MST-Kruskal( $G,w$ ) utilizza le procedure

- Make-Set, Find-Set, Union
- Make-Set, Delete-Set
- Find-Arc, Insert-Arc
- Make-Set, Find-Arc, Compute-Cost
- Make-Set, Union, Compute-Cost

224. L'algoritmo MST-Kruskal( $G,w$ )

- trova l'albero dei cammini minimi del grafo  $G$
- trova l'albero di copertura minima del grafo  $G$
- trova un cammino minimo del grafo  $G$
- trova tutti gli alberi di copertura del grafo  $G$
- costruisce un albero che copre tutti i vertici in  $V$

225. L'algoritmo MST-Kruskal( $G,w$ ) costruisce un MST

- scegliendo sempre l'arco di costo minimo fra quelli non inseriti
- scegliendo sempre l'arco di costo minimo fra quelli non inseriti che non chiuda un anello
- scegliendo sempre l'arco di costo minimo fra i non inseriti che non chiuda un anello e che sia connesso al sottoalbero corrente
- scegliendo sempre l'arco di costo minimo che non chiuda un anello
- scegliendo sempre l'arco di costo massimo fra quelli non inseriti che non chiuda un anello

226. L'algoritmo MST-Kruskal( $G,w$ )

- mantiene gli archi di  $G$  ordinati per costo decrescente
- mantiene gli archi di  $G$  ordinati per costo non crescente
- mantiene gli archi di  $G$  ordinati per costo non decrescente
- mantiene gli archi di  $G$  ordinati per costo crescente
- non necessita di particolari ordinamenti degli archi di  $G$

227. Per provare la correttezza dell'algoritmo di kruskal applicato a un grafo  $G=(V,E)$  si considera una partizione del grafo  $G'=(V,\bar{E})$  corrente in due componenti non connesse  $G_1$  e  $G_2$  e si prova che

- l'unione di  $G_1$  e  $G_2$  fornisce l'albero di copertura cercato
- $(V, \bar{E} \cup e)$ , e arco di costo minimo che unisce  $G_1$  e  $G_2$ , è un sottografo di qualche MST
- il grafo  $G' \setminus G_1 \cup G_2$  è un sottografo della MST cercata
- $(V, \bar{E} \cup e)$ , e arco di costo minimo che unisce  $G_1$  e  $G_2$ , è un MST
- $(V, \bar{E} \cup e)$ , e arco di costo minimo che unisce  $G_1$  e  $G_2$ , è un sottografo di  $G$

228. La complessità in tempo dell'algoritmo di kruskal è

- $O(V) + O(E \log E)$
- $O(E \log^* E)$
- $\Theta(V \log E)$
- $\Theta(E \log V)$
- $O(E \log E)$

229. L'algoritmo MST-Prim( $G,w,r$ ) utilizza le procedure

- Make-Set, Find-Set, Union
- Extract-Min
- Find-Arc, Extract-Min
- Make-Set, Find-Arc, Compute-Cost
- Make-Set, Union, Compute-Cost

230. L'algoritmo MST-Prim( $G,w,r$ )

- trova l'albero dei cammini minimi del grafo  $G$
- trova l'albero di copertura minimo del grafo  $G$
- trova un cammino minimo del grafo  $G$
- trova tutti gli alberi di copertura del grafo  $G$
- costruisce un albero che copre tutti i vertici in  $V$

231. L'algoritmo MST-Prim( $G,w,r$ )

- costruisce un MST scegliendo sempre l'arco di costo minimo fra quelli non inseriti
- costruisce un MST scegliendo sempre l'arco di costo minimo fra quelli non inseriti che non chiuda un anello
- costruisce un MST scegliendo sempre l'arco di costo minimo fra i non inseriti che non chiuda un anello e che sia connesso al sottoalbero corrente
- costruisce un MST scegliendo sempre l'arco di costo minimo che non chiuda un anello
- costruisce un MST scegliendo sempre l'arco di costo massimo fra quelli non inseriti che non chiuda un anello

232. L'algoritmo MST-Prim( $G,w,r$ )

- mantiene gli archi di  $G$  ordinati per costo decrescente
- mantiene gli archi di  $G$  ordinati per costo non crescente
- mantiene gli archi di  $G$  ordinati per costo non decrescente
- mantiene gli archi di  $G$  ordinati per costo crescente
- non necessita di particolari ordinamenti delegando alla Extract-Min la gestione dei costi degli archi di  $G$

233. La complessità in tempo dell'algoritmo di Prim è

- $O(E + V \log V)$
- $O(E \log^* E)$
- $\Theta(V \log E)$
- $\Theta(E \log V)$
- $O(E \log E)$

234. Il problema dei cammini minimi con sorgente singola, dato un grafo  $G=(V,E)$  con pesi  $w:E \rightarrow R$  e un vertice  $s \in V$ , richiede di trovare

- per ogni arco  $e \in E$  il cammino di peso minimo originato in  $s$  e passante per  $e$
- per ogni vertice  $v \in V$  il cammino di peso minimo da  $s$  a  $v$
- per ogni peso  $w$  il cammino più breve di peso  $w$  originato in  $s$
- per ogni vertice  $v \in V$  tutti i cammini originati in  $s$  che terminano in  $v$
- per ogni CFC di  $G$ , il cammino di peso minimo che connette  $s$  alla CFC

235. Il problema dell'individuazione dei cammini minimi con sorgente singola su un grafo  $G=(V,E)$  può essere esteso a comprendere

- l'individuazione del cammino minimo fra una coppia di vertici  $u$  e  $v$
- l'individuazione del cammino minimo fra due CFC
- l'individuazione del cammino minimo fra ogni coppia di archi in  $E$
- l'individuazione del cammino minimo fra una sorgente  $s \in V$  ed ogni altro vertice in  $V$
- l'individuazione dei cammini minimi fra tutte le coppie di vertici in  $V$

236. L'individuazione dei cammini minimi con sorgente singola sul grafo  $G=(V,E)$  può essere effettuata sotto l'ipotesi che

- non esistano in  $G$  componenti fortemente connesse di peso negativo
- non esistano in  $G$  archi di peso negativo
- non esistano in  $G$  vertici di peso negativo
- non esistano in  $G$  cicli di peso negativo
- non esistano in  $G$  cammini di peso negativo

237. La rappresentazione interna dei cammini nel problema dell'individuazione dei cammini minimi con sorgente singola è analoga a quella

- delle CFC
- delle heap di Fibonacci
- delle heap binomiali
- degli MST
- degli alberi BFS

238. Nella rappresentazione interna dei cammini minimi, per ogni vertice  $v \in V$  viene mantenuto

- un predecessore  $\pi(v)$
- un figlio  $\phi(v)$
- un puntatore alla sorgente  $\sigma(v)$
- un grafo  $G(v)$
- non so

239. L'individuazione dei cammini minimi con sorgente singola  $s$  sul grafo  $G=(V,E)$  comporta

l'individuazione di un albero dei cammini minimi  $G\pi=(V\pi, E\pi)$  in cui

- $V\pi$  è l'insieme dei vertici raggiungibili da  $s$  in  $G$
- per ogni  $v \in V\pi$  l'unico cammino semplice da  $s$  a  $v$  in  $G\pi$  è il cammino minimo da  $s$  a  $v$  in  $G$
- per ogni  $v \in V\pi$  l'unico cammino semplice da  $s$  a  $v$  in  $G\pi$  è un cammino minimo da  $s$  a  $v$  in  $G$
- $G\pi$  forma un albero con radice in  $v$
- per ogni  $e \in E\pi$  l'unico cammino semplice da  $s$  a  $v$  in  $G\pi$  è il cammino minimo passante per  $e$

240. L'algoritmo InitializeSingleSource, in Dijkstra:

- inizializza l'array contenente i dati del rafforzamento del costo del cammino minimo da  $s$  a ogni  $v \in V$
- inizializza l'array contenente i dati del rilassamento del costo del cammino minimo dalla sorgente  $s$  ad ogni vertice  $v$
- inizializza variabili  $d[v]$  contenenti, per ogni  $v$ , una stima del costo del cammino minimo espressa come un suo limite superiore
- inizializza a 0 l'array  $d[v]$ , per ogni  $v \in V$ , e le variabili poi usate in Dijkstra
- inizializza le variabili relative alla sorgente  $s$

241. L'algoritmo Relax, in Dijkstra

- per un arco  $(u,v)$  verifica se è possibile migliorare il cammino minimo per  $v$  passante per  $u$
- per un vertice  $v$  verifica se è possibile rilassare la stima del cammino di costo minimo che arriva a  $v$
- per la sorgente  $s$ , verifica se è possibile rilassare il cammino di costo minimo da  $s$  ad ogni  $v \in V$
- per il grafo  $G=(V,E)$ , verifica se è possibile identificare un rilassamento dei costi dei cammini minimi da  $s$
- per il grafo  $G=(V,E)$ , verifica se è possibile identificare un rilassamento del costo del cammino minimo da  $s$  a  $v \in V$

242. L'algoritmo di Dijkstra, sorgente singola  $s$

- percorre in maniera BFS il grafo  $G$  a partire dal vertice  $s$
- percorre in maniera DFS il grafo  $G$  a partire dal vertice  $s$
- associa ad ogni vertice  $v \in V$  una stima del rilassamento dell'arco  $(s,v)$  che porta in  $v$
- aggiorna ricorsivamente i puntatori del cammino minimo da  $s$  ad ogni  $v \in V$
- mantiene un insieme  $S$  di vertici  $v$  per cui il costo del cammino minimo da  $s$  a  $v$  è già stato determinato

243. La correttezza dell'algoritmo di Dijkstra, applicato ad un grafo orientato e pesato  $G=(V,E)$  e sorgente  $s$ , si prova dimostrando che

- l'algoritmo termina in un tempo di CPU che è  $O(V^2)$
- al termine dell'esecuzione è possibile raggiungere ogni vertice  $v \in V$  a partire da  $s$
- al termine dell'esecuzione l'algoritmo ha individuato ciò che doveva individuare
- al termine dell'esecuzione si ha  $d[u] = \delta(s,u)$  per ogni  $u \in V$
- l'asserto della correttezza di Dijkstra è rispettato per ogni possibile  $s \in V$

244. La complessità computazionale dell'algoritmo di Dijkstra è

- $O(E^2)$
- $O(V^2 + E)$
- $O(V \log E)$
- $O(V^2)$
- $O(V^2 \log E)$

245. L'algoritmo di Bellman - Ford risolve il problema

- dell'albero di copertura minima di un grafo
- dei cammini minimi fra tutte le coppie di vertici di un grafo
- dei cammini minimi da una sorgente ad ogni altro nodo di un grafo
- dei cammini minimi da una sorgente ad ogni altro nodo di un grafo, accettando archi di costo negativo
- dell'individuazione delle componenti connesse di un grafo

246. L'algoritmo di Bellman - Ford

- è applicabile quando tutti gli archi del grafo hanno costo negativo
- permette di individuare cammini minimi anche in un grafo con archi di costo negativo
- restituisce false se esistono cicli di costo negativo e true altrimenti
- restituisce un identificativo di "nessuna soluzione" nel caso esistano cicli di costo negativo
- è una versione meno efficiente dell'algoritmo di Dijkstra

247. L'algoritmo di Bellman - Ford

- si basa sulla stessa funzione Relax usata anche da Dijkstra
- permette di individuare l'albero di copertura minima anche in grafi con archi di costo negativo
- permette di individuare in modo deterministico i cammini minimi in un grafo, senza rilassamenti o approssimazioni
- individua in modo randomizzato i cammini minimi in un grafo
- è una procedura ricorsiva per l'individuazione dei cammini minimi

248. L'algoritmo di Bellmann-Ford applicato a un grafo  $G=(V,E)$  restituisce

- true se  $G$  non contiene archi di costo negativo, false altrimenti
- l'albero dei cammini minimi da  $s$  a ogni  $v$  in  $V$
- true se  $G$  non contiene archi di costo negativo, nel qual caso si ha che  $d[v]=\delta(s,v)$  per ogni  $v \in V$ , false altrimenti
- false se ci sono archi negativi nel grafo, true se  $G$  non contiene archi di costo negativo, nel qual caso si ha che  $d[v]=\delta(s,v)$  per ogni  $v \in V$
- l'albero di copertura minima del grafo  $G$

249. La complessità dell'algoritmo di Bellmann - Ford è

- $O(V E)$
- $O(V+E)$
- $O(V \log E)$
- $O(E \log V)$
- $O(V^2)$

250. Nel caso di un DAG è possibile calcolare i cammini minimi sfruttando

- un ordinamento lineare dei vertici di G
- un ordinamento quadratico dei vertici di G
- un ordinamento crescente dei vertici di G
- un ordinamento topologico dei vertici di G
- un ordinamento binario dei vertici di G

251. La complessità di DAG-shortest-path è

- $O(V E)$
- $O(V \log E)$
- $O(E \log V)$
- $O(V^2)$
- $O(V+E)$

252. Il problema dei cammini minimi fra tutte le coppie, dato un grafo pesato  $G=(V,E,W)$ , richiede di

- trovare per ogni coppia  $u,v \in V$  il minimo costo di un cammino da  $u$  a  $v$
- trovare per ogni coppia  $u,v \in E$  il minimo costo di un cammino da  $u$  a  $v$
- trovare per ogni coppia  $u,v \in V$  il minimo costo di un cammino da  $s$  a  $v$  passante per  $u$
- trovare per ogni coppia  $u,v \in V$  il minimo costo di un cammino da  $s$  a  $u$  passante per  $v$
- trovare per ogni arco  $(u,v) \in E$  il minimo costo di un cammino da  $s$  passante per  $(u,v)$

253. La matrice dei predecessori  $\pi=\{\pi_{uv}\}$  calcolata assieme ai cammini minimi fra tutte le coppie di un grafo  $G=(V,E)$  è tale per cui

- $\pi_{uv}$  è NIL se non c'è un cammino da  $u$  a  $v$ , altrimenti è un puntatore a un predecessore di  $v$  su di un cammino minimo da  $u$
- $\pi_{uv}$  è NIL se  $u=v$ , altrimenti è un puntatore a un predecessore di  $v$  su di un cammino minimo da  $u$
- $\pi_{uv}$  è NIL se  $u=v$  o se non c'è un cammino da  $u$  a  $v$ , altrimenti è un predecessore di  $v$  su di un cammino minimo da  $u$
- $\pi_{uv}$  è NIL se  $u=v$  o se non c'è un cammino da  $u$  a  $v$ , altrimenti è un predecessore di  $v$  su di un cammino minimo in  $G$

254. La riga  $i$ -esima della matrice dei predecessori identifica

- un albero dei cammini minimi con radice in  $i$
- tutti i cammini minimi che conducono a  $i$
- tutti i cammini minimi passanti per  $i$
- tutti gli alberi di copertura contenenti  $i$
- un albero di copertura minimo radicato in  $i$

255. L'algoritmo di Floyd-Warshall risolve il problema

- dell'albero di copertura minimo di un grafo
- dei cammini minimi fra tutte le coppie di vertici di un grafo
- dei cammini minimi da una sorgente ad ogni altro nodo di un grafo
- dei cammini minimi da una sorgente ad ogni altro nodo, accettando archi di costo negativo
- dell'individuazione delle componenti connesse di un grafo

256. L'algoritmo di Floyd-Warshall

- è un algoritmo di programmazione dinamica
- è un algoritmo greedy
- è un algoritmo approssimato
- è un algoritmo euristico
- è un algoritmo ottimo

257. L'algoritmo di Floyd-Warshall

- accetta cicli di costo negativo
- assume che tutti gli archi abbiano costo non negativo
- assume che tutti gli archi abbiano costo non positivo
- non fa assunzioni sui costi degli archi
- accetta archi di peso negativo ma non cicli negativi

258. Il passo ricorsivo implementato da Floyd-Warshall per estendere un cammino minimo da s a t passando solo per i nodi  $v_1, \dots, v_i$  è:

- $d_{st}(k) = \min(d_{st}(k-1), d_{sk}(k-1))$  (se  $k > 0$ )
- $d_{st}(k) = d_{sk}(k-1) + d_{kt}(k-1)$  (se  $k > 0$ )
- $d_{st}(k) = \min(d_{st}(k-1), d_{sk}(k-1) + d_{kt}(k-1))$  (se  $k > 0$ )
- $d_{st}(k) = \min(d_{st}(k-1), d_{sk}(k))$  (se  $k > 0$ )
- non c'è nessun passo ricorsivo

259. L'algoritmo di Floyd-Warshall viene realizzato

- con una chiamata ricorsiva a Floyd-Warshall(W)
- con tre cicli for innestati
- Basandosi sull'algoritmo Relax
- con due cicli while innestati
- in pseudocodice

260. La complessità computazionale dell'algoritmo di Floyd-Warshall applicato a un grafo  $G=(V,E)$  è

- $O(V^3)$
- $\Theta(V^3)$
- $O(V^2)$
- $O(V E)$
- $O(V \log E)$

261. I problemi decisionali sono la classe di problemi dove per ogni possibile ingresso un algoritmo deve

- trovare la corrispondente soluzione di costo minimo
- risolvere il problema
- computare un'uscita corrispondente alla stringa di input ricevuta in ingresso
- scegliere una di due risposte possibili: "si" o "no"
- individuare la decisione che risolve il problema

262. La classe delle funzioni corrispondenti a problemi decisionali è quella delle funzioni

- computabili del tipo  $f: N \rightarrow \{0,1\}$
- $f: N \rightarrow \{0,1\}$
- computabili del tipo  $f: N \rightarrow R$
- computabili del tipo  $f: \text{stringhe} \rightarrow \{\text{"si"}, \text{"no"}\}$
- le funzioni non c'entrano niente

263. Il problema del sottografo completo richiede:

- Dato un grafo  $G$ , stabilire se  $G$  contiene un sottografo completo
- Dati un grafo  $G$  di  $n$  vertici, stabilire se  $G$  contiene un sottografo completo
- Dati un grafo  $G$  e un intero  $n$ , stabilire se il grafo  $G$  contiene un sottografo completo con  $n$  vertici
- Dati un grafo  $G$  di  $n$  vertici, stabilire se  $G$  contiene un sottografo completo con  $n$  archi

264. Il problema del cammino hamiltoniano per un grafo  $G$  richiede:

- stabilire se esiste un ciclo che tocchi tutti i vertici di  $G$  una e una sola volta
- stabilire se esiste un cammino che tocchi tutti i vertici di  $G$  una e una sola volta tornando alla fine al vertice di partenza
- stabilire se esiste un cammino che tocchi tutti i vertici di  $G$
- stabilire se esiste un cammino elementare che tocchi tutti i vertici di  $G$
- stabilire se esiste un cammino che tocchi tutti i vertici di  $G$  una e una sola volta

265. Il problema del cammino euleriano, dato un grafo  $G$ , richiede di stabilire:

- se esiste un cammino che tocchi tutti i vertici di  $G$  una e una sola volta
- se esiste un cammino elementare che percorra tutti gli archi di  $G$  una e una sola volta
- se esiste un cammino che percorra tutti gli archi di  $G$  una e una sola volta
- se esiste un cammino che percorra tutti gli archi di  $G$

266. Il problema SAT, data una  $k$ -CNF  $F$ , richiede di stabilire:

- se  $F$  è soddisfacibile, cioè se esiste un assegnamento di valori 0 e 1 alle variabili in  $F$  per cui il valore dei disgiunti diventi 1
- se  $F$  è soddisfacibile, cioè se esiste un assegnamento di valori 0 e 1 alle variabili in  $F$  per cui il valore di  $F$  diventi 1
- se  $F$  è soddisfacibile, cioè se esiste un assegnamento ammissibile di valori 0 e 1 alle variabili in  $F$
- se  $F$  è soddisfacibile, cioè se esiste un assegnamento di valori 0 e 1 alle variabili in  $F$  per cui  $F$  è soddisfatta

267. Una  $k$ -CNF è

- una CNF definita su  $k$  congiunti
- un insieme di  $k$  CNF ognuna singolarmente soddisfacibile
- la disgiunzione di  $k$  CNF
- una CNF in cui ogni congiunto ha  $k$  termini
- non ricordo

268. Un problema di ottimizzazione richiede

- di individuare quale fra "si" o "no" sia la soluzione del problema
- di trovare la soluzione ottima per il problema dato
- di trovare il massimo o il minimo di una funzione
- di utilizzare in modo ottimo le risorse disponibili
- di ottimizzare i dati del problema

269. Dato un problema di ottimizzazione PO e il corrispondente problema decisionale PD

- PO e PD hanno la stessa complessità
- PD è computazionalmente più complesso di PO
- PO è computazionalmente più complesso di PD
- è possibile ricondurre PO a PD tramite una ricerca binomiale su una soglia sul costo ottimo
- non esiste sempre un problema decisionale corrispondente ad un problema di ottimizzazione

270. Una istanza di un problema

- è una specifica della struttura di come appare
- è la specifica della modalità con cui passare il problema ad un algoritmo
- è una implementazione del problema
- è un suo caso particolare in cui vengono specificati tutti i suoi elementi costitutivi
- è una stringa binaria

271. Per codifica di un problema si intende:

- la corrispondenza fra l'insieme delle istanze del problema e un insieme di stringhe binarie:  $e: I \rightarrow \{0,1\}^*$
- corrispondenza fra il problema e l'insieme delle sue istanze
- la modalità con cui si passa il problema all'algoritmo risolutivo
- l'individuazione di un opportuno schema biettivo di corrispondenza fra gli elementi del problema e le strutture sintattiche rappresentative
- l'implementazione del problema

272. Un problema decisionale PD è nella classe P

- se esiste un algoritmo che risolve un'istanza del problema PD in tempo polinomiale
- se esiste un algoritmo che risolve qualsiasi istanza del problema PD in tempo polinomiale
- se è possibile risolvere qualsiasi istanza del problema PD in tempo polinomiale
- se esiste un algoritmo che risolve qualsiasi istanza del problema PD in tempo non deterministico polinomiale
- se è facile da risolvere

273. Una funzione  $f: \{0,1\}^* \rightarrow \{0,1\}^*$  è calcolabile in tempo polinomiale

- se esiste un algoritmo polinomiale che, dato in input un qualsiasi  $x = \{0,1\}^*$ , produce come output  $\{0,1\}^*$
- se esiste un algoritmo che, dato in input un qualsiasi  $x = \{0,1\}^*$ , produce come output "si" in tempo polinomiale
- se esiste un algoritmo che, dato in input un qualsiasi  $x = \{0,1\}^*$ , produce come output "no" in tempo polinomiale

- se esiste un algoritmo polinomiale che, dato in input un qualsiasi  $x=\{0,1\}^*$ , produce come output  $f(x)$ .
- se il tempo di calcolo è del tipo  $nx$

274. Un problema decisionale PD è nella classe NP

- se non esiste un algoritmo che risolve qualsiasi istanza di PD in tempo polinomiale (rispetto alla dimensione della istanza).
- se esiste un algoritmo che risolve qualsiasi istanza di PD in tempo non polinomiale (rispetto alla dimensione della istanza).
- se esiste un algoritmo approssimato che risolve qualsiasi istanza di PD in tempo polinomiale (rispetto alla dimensione della istanza).
- se esiste un algoritmo deterministico che risolve qualsiasi istanza di PD in tempo non polinomiale (rispetto alla dimensione della istanza).
- se esiste un algoritmo non-deterministico che risolve qualsiasi istanza di PD in tempo polinomiale (rispetto alla dimensione della istanza).

275. Un algoritmo non deterministico è un programma

- che può anche contenere istruzioni del tipo  $\text{goto } \{L_1, \dots, L_n\}$
- che può dare risultati diversi ad ogni esecuzione
- che risolve problemi NP
- che contiene chiamate alla funzione `random()`
- che non è definito completamente

276. Per realizzazione di un algoritmo non deterministico A si intende

- l'implementazione dell'algoritmo A
- l'individuazione dello pseudocodice di A
- ciascuna delle possibili diverse esecuzioni di A
- l'applicazione di A ad un'istanza del problema che risolve

277. Un algoritmo non deterministico A calcola una funzione  $f : N \rightarrow \{0,1\}$  se:

- per ogni  $a \in N$  tale che  $f(a)=1$  tutte le realizzazioni di A terminano
- per ogni  $a \in N$  tale che  $f(a)=1$  tutte le realizzazioni di A terminano restituendo 1
- per ogni  $a \in N$  tale che  $f(a)=1$  esiste una realizzazione di A che ritorna 1
- per ogni  $a \in N$  tale che  $f(a)=0$  esiste una realizzazione di A che ritorna 0
- per ogni  $a \in N$  tale che  $f(a)=0$  tutte le realizzazioni di A terminano restituendo 0

278. Per il problema del ciclo euleriano esiste un algoritmo risolutore:

- polinomiale
- lineare
- non deterministico polinomiale
- logaritmico
- non deterministico lineare

279. Si sa che

- $NP \subseteq P$
- $P \subseteq NP$
- $P = NP$
- $P \neq NP$
- $P \subset NP$

280.  $f : N \rightarrow \{0, 1\}$  è riducibile polinomialmente a  $g : N \rightarrow \{0, 1\}$  se

- esiste una funzione  $h$  tale che per ogni  $x$ :  $f(x) = g(h(x))$
- esiste una funzione  $h$ , calcolabile in tempo polinomiale, tale che per ogni  $x$ :  $g(x) = f(h(x))$
- esiste una funzione  $h$ , calcolabile in tempo polinomiale, tale che per ogni  $x$ :  $f(x) = h(g(x))$
- esiste una funzione  $h$ , calcolabile in tempo polinomiale, tale che per ogni  $x$ :  $f(x) = g(h(x))$
- esiste una funzione  $h$ , calcolabile in tempo polinomiale, tale che per ogni  $x$ :  $h(x) = g(f(x))$

281.  $f : N \rightarrow \{0, 1\}$  è NP-completo se e solo se:

- non è risolvibile in tempo polinomiale
- $f \in NP$
- per ogni  $g \in NP$ ,  $g$  è riducibile polinomialmente a  $f$
- non esistono algoritmi, neanche non deterministici, che risolvono  $f$  in tempo polinomiale
- $f$  non è in  $P$

282. Per dimostrare la NP completezza di una funzione  $f$  dalla definizione di NP completezza si richiede di:

- dimostrare che  $f$  è non deterministica polinomiale
- dimostrare che  $f$  non è in  $P$
- dimostrare che  $f$  è riducibile polinomialmente qualunque altra funzione in  $NP$
- che la funzione è in  $NP$
- dimostrare che qualunque altra funzione in  $NP$  è riducibile polinomialmente alla funzione data

283. Per dimostrare la NP completezza di una funzione  $f$  è necessario

- mostrare che la funzione  $f$  è in  $NP$
- ridurre polinomialmente  $f$  a  $g$ , per ogni  $g \in NP$
- mostrare che  $g \leq_p f$  per qualche problema  $g$  che è già noto essere NP completo
- mostrare che  $f$  non è in  $P$

284. La prova di NP completezza del problema del sottografo completo (CSP) può essere fatta

- riducendo SAT a CSP
- riducendo CSP a SAT
- riducendo CSP a TSP
- aumentando SAT a CSP
- aumentando CSP a SAT

285. Il problema vertex cover, dato un grafo  $G=(V,E)$ , richiede di:

- trovare il massimo numero di archi che abbiano un estremo in uno dei vertici in  $V$
- trovare il minimo numero di archi tale per cui ogni vertice in  $V$  abbia almeno un arco uscente
- trovare un sottinsieme  $S$  di dimensione minima dei vertici in  $V$ , tale per cui ogni arco abbia un vertice in  $S$
- trovare il massimo numero di vertici in  $V$  tale per cui gli archi restino coperti
- coprire tutti i vertici di  $G$  con un cammino hamiltoniano

286. Una rete di flusso è un grafo

- in cui ogni arco ha associato una capacità, cioè un peso non negativo
- orientato pesato  $G=(V,A,c)$ , con due nodi particolari: una sorgente  $s$  e un pozzo  $t$
- diretto, aciclico e连通的
- orientato pesato che può anche contenere cicli di costo negativo

287. una rete di flusso può modellare:

- reti di comunicazione
- reti da pesca
- circuiti elettrici
- reti idrauliche

288. Un flusso ammissibile per  $G$  è

- Una funzione fra archi di  $G$  e  $R$  che soddisfa la capacità degli archi e la conservazione ai nodi
- Una funzione fra archi di  $G$  e  $R$  che soddisfa la conservazione degli archi e la capacità ai nodi
- Una funzione fra vertici di  $G$  e  $R$  che soddisfa la capacità degli archi e la conservazione ai nodi
- Una funzione fra vertici di  $G$  e  $R$  che soddisfa la conservazione degli archi e la capacità ai nodi

289. Il problema del flusso massimo (max flow) chiede di

- determinare la sorgente  $s$  da cui inviare il massimo flusso ammissibile a un pozzo  $t$ .
- determinare il valore del massimo flusso ammissibile inviabile da una sorgente  $s$  a un pozzo  $t$ .
- determinare il valore del massimo flusso inviabile da una sorgente  $s$  a un pozzo  $t$ .
- determinare il costo del massimo flusso ammissibile inviabile da una sorgente  $s$  a un pozzo  $t$ .

290. Una rete di flusso può diventare una rete di circolazione

- aggiungendo un arco  $(s,t)$  con capacità infinita e richiedendo che il flusso  $f(ts,t)$  sia il più grande possibile.
- aggiungendo un arco  $(t,s)$  con capacità pari al flusso massimo.
- aggiungendo un arco  $(t,s)$  con capacità infinita e richiedendo che il flusso  $f(t,s)$  sia il più grande possibile.
- aggiungendo un arco  $(t,s)$  con capacità infinita e richiedendo che il flusso  $f(t,s)$  sia il più breve possibile.

291. In un taglio  $(A,B)$  di una rete di flusso  $G = (V, E)$  con sorgente  $s$  e pozzo  $t$  in cui

- $s \in A$  e  $t \in A$
- $s \in A$  e  $t \in B$
- $s \in B$  e  $t \in B$
- $s$  e  $t$  possono appartenere a qualunque delle due partizioni

292. In un taglio (A,B) di una rete di flusso  $G = (V, E)$  con sorgente s e pozzo t ha una capacità  $c(A,B)$

- è pari alla somma delle capacità degli archi con il primo estremo in B e il secondo in A
- è pari alla somma delle capacità degli archi della partizione A
- è pari alla somma delle capacità degli archi della partizione B
- è pari alla somma delle capacità degli archi con il primo estremo in A e il secondo in B
- è pari alla somma delle capacità di tutti gli archi del grafo

293. In un taglio (A,B) di una rete di flusso  $G = (V, E)$  con sorgente s e pozzo t il flusso attraverso il taglio

- è pari alla somma dei flussi sugli archi con il primo estremo in B e il secondo in A
- è pari alla somma dei flussi sugli archi della partizione A
- è pari alla somma dei flussi sugli archi della partizione B
- è pari alla somma dei flussi su tutti gli archi del grafo
- è pari alla somma dei flussi sugli archi con il primo estremo in A e il secondo in B

294. Il teorema max flow – min cut per reti di flusso asserisce che

- il flusso minimo in una rete  $G=(V,A)$  è pari alla capacità del taglio di G di capacità minima.
- il flusso massimo in una rete  $G=(V,A)$  è pari alla capacità del taglio di G di capacità minima.
- il flusso massimo in una rete  $G=(V,A)$  è pari alla capacità del taglio di G di capacità massima.
- il flusso minimo in una rete  $G=(V,A)$  è pari alla capacità del taglio di G di capacità minima.

295. Data una rete  $G(V,A)$  su cui circola un flusso  $f$ , il grafo residuo  $G_f(V,A_f)$

- è un sottografo di G contenente solo gli archi di A con capacità residua positiva o nulla
- è un supergrafo di G contenente gli archi di A e gli archi residui
- è un sottografo di G contenente solo gli archi di A con capacità residua strettamente positiva
- è un supergrafo di G contenente gli archi di A con capacità residua incrementale

296. Un cammino aumentante nella rete G in cui circola un flusso f (eventualmente nullo)

- è un cammino da s a t aumentato di un arco
- è un cammino fra due nodi qualunque del grafo residuo che permette di aumentare il flusso
- è un cammino fra due nodi qualunque del grafo G che permette di aumentare il flusso
- è un cammino da s a t nel suo grafo residuo

297. Il flusso lungo un cammino aumentante può essere aumentato

- al massimo di un quantitativo pari alla minima capacità residua  $c_f$  degli archi del cammino
- al massimo di un quantitativo pari alla massima capacità residua  $c_f$  degli archi del taglio minimo
- al minimo di un quantitativo pari alla capacità residua  $c_f$  degli archi del cammino
- al massimo di un quantitativo pari alla massima capacità residua  $c_f$  degli archi del cammino

298. L'algoritmo di Ford–Fulkerson a cammini aumentanti

- Permette di ridurre flussi in certi archi se si accorge di scelte sbagliate
- Individua una successione di cammini aumentanti nei grafi residui
- Aumenta sempre i flussi ammissibili che via via individua
- Individua cicli di flussi di costo negativo

299. La capacità residua di un arco  $(u, v)$  con flusso  $f(u, v)$  è

- $c_f(u, v) = f(u, v) - c(u, v)$  direzione  $u \rightarrow v$
- $c_f(u, v) = c(u, v) - f(u, v)$  direzione  $u \rightarrow v$
- $c_f(u, v) = f(v, u)$  direzione  $u \rightarrow v$
- $c_f(u, v) = f(v, u)$  direzione  $v \rightarrow u$

300. La complessità dell'algoritmo di Ford-Fulkerson è

- Non polinomiale
- $O(|E| V)$
- $O(|E| f^*)$ , dove  $f^*$  è il valore del flusso massimo
- $O(|E| \log |E|)$

301. L'algoritmo di Edmonds-Karp per flussi massimi in reti di flusso

- Inserisce un passo di breadth first search in Ford-Fulkerson
- Utilizza un approccio diverso da quello seguito da Ford-Fulkerson
- Si basa su Ford-Fulkerson e richiede di individuare solo cammini aumentanti con numero minimo di archi
- Estende alle reti di flusso l'algoritmo di Kruskal

302. La complessità di Edmonds-Karp è

- $O(VE)$
- $O(EV^2)$
- $O(VE^2)$
- $O(V f^*)$ , dove  $f^*$  è il valore del flusso massimo

303. Il problema min-cost max-flow (flusso massimo a costo minimo) richiede di

- Trovare fra tutti i flussi a costo minimo quello a flusso massimo
- Trovare il flusso massimo con il minimo costo computazionale
- Trovare fra tutti i flussi massimi quello a costo minimo
- Trovare fra tutti i flussi computazionali quello a minimo costo del cammino aumentante

304. Casi particolari del problema min-cost max-flow includono

- Il problema dell'albero di copertura massimo a costo minimo
- Il problema del cammino a costo minimo e dei k cammini disgiunti
- Il problema del cammino massimo a costo minimo
- Il problema del cammino a costo e dei k alberi di copertura disgiunti

305. Nel problema min-cost max-flow gli archi residui  $(u,v)$  in una rete con flusso  $f$

- hanno capacità invariate e  $\text{cost}_f(uv) = \text{cost}(uv)$ , e  $\text{cost}_f(vu) = -\text{cost}(uv)$ .
- hanno capacità invariate e  $\text{cost}_f(vu) = \text{cost}(uv)$ , e  $\text{cost}_f(uv) = -\text{cost}(uv)$ .
- hanno capacità definite come nel flusso massimo e  $\text{cost}_f(uv) = \text{cost}(uv)$ , e  $\text{cost}_f(vu) = -\text{cost}(uv)$ .
- hanno capacità definite come in Kruskal e  $\text{cost}_f(uv) = d(uv)$ , e  $\text{cost}_f(vu) = -d(uv)$ .

306. Nell'algoritmo cycle cancelling per il problema min-cost max-flow

- Il flusso s-t viene incrementalmente aumentato su cammini aumentanti di costo decrescente
- Il flusso s-t viene incrementalmente aumentato individuando cicli di flussi da cancellare
- Il flusso s-t viene incrementalmente aumentato analogamente ad Edmonds-Karp
- Dopo la prima allocazione la quantità di flusso da s a t non cambia mai

