Lösningsförslag exempel – inspera 2024

1. Se inspera2024.py
2. Dynamisk lista
    a. En dynamisk lista består av länkar som var och en håller ett värde/ element ur listan och en referens till nästa länk i listan. Själva listan håller en referens till den första länken, eller None om listan är tom.
    b. linked_list_function skapar en ny länk som håller det data som skickas in till funktionen. Den refererar det som var första länken i listan innan funktionsanropet som nästa länk i kedjan. Den nya länken sätts in som listans första. (Eller enklare: linked_list_function sätter in det nya värdet data först i listan.)
    c. Tidskomplexiteten är konstant, vi behöver inte gå igenom listans länkar för att sätta in en ny länk i början.
3. Sortering
    a. Innehållet i sort_arr blir [2, 3, 4, 6, 1, 8, 5, 0]
    (Längre förklaring:
    Vi har som in-argument sort_arr = [3, 6, 2, 4, 1, 8, 5, 0], lo=0, mid=2, hi=4 och tmp_arr=[0, 0, 0, 0, 0, 0, 0, 0]
    Vi sätter
    n = 0 (lo), idx_lo = 0 (lo) och idx_hi = 2 (mid)
    Villkoret i while-satsen är att idx_lo < mid och idx_hi < hi,
    Eftersom 0 (idx_lo) < 2 (mid) och 2 (idx_hi) < 4 (hi) så går vi in i while-loopen.
    Här sätter vi item_lo = sort_arr[0] = 3 och
    item_hi = sort_arr[2] = 2
    Vi kollar om item_lo < item_hi, dvs om 3 < 2. Det är det inte, så vi går till else-satsen. Här sätter vi tmp_arr[0] = 2 (item_hi) och ökar på idx_hi till 3. På raden efter if-satsen ökar vi n till 1.
    Då är vi klara med ett varv i den övre while-loopen och kontrollerar villkoret. Vi har fortfarande idx_lo < mid (0<2), och idx_hi < hi (3<4), så vi går ännu ett varv.
    item_lo blir 3 igen och item_hi blir sort_arr[3]=4
    Vi kollar om item_lo < item_hi, dvs om 3 < 4. Det är det, så vi går in i if-satsen, sätter tmp_arr[1] = 3 och ökar på idx_lo till 1. På raden efter if-satsen ökar vi n till 2.
    Då är vi klara med det andra varvet i den övre while-loopen och kontrollerar villkoret. Vi har fortfarande idx_lo < mid (1<2), och idx_hi < hi (3<4), så vi går ännu ett varv.
    item_lo blir sort_arr[1]=6 och item_hi blir 4 igen.

Vi kollar om item_lo < item_hi, dvs om 6 < 4. Det är det inte, så vi går in i else-satsen. Vi sätter tmp_arr[2]=4 och ökar på idx_hi till 4. På raden efter if-satsen ökar vi n till 3.
Då är vi klara med det tredje varvet i den övre while-loopen och kontrollerar villkoret. Vi har fortfarande idx_lo < mid (1<2), men idx_hi är inte mindre än hi, de är båda 4. Då avslutar vi den första while-loopen. Här är sort_arr oförändrad mot hur den kom in,, lo, mid och hi har inte ändrats. n =3, idx_lo=1, idx_hi=4 och tmp_arr=[2, 3, 4, 0, 0, 0, 0, 0].
Villkoret till nästa while-loop är idx_lo < mid, dvs 1<2, som stämmer, så vi går in i loopen. Vi sätter tmp_arr[3] = sort_arr[1] = 6. Vi ökar på idx_lo till 2 och n till 4. Till nästa varv är inte villkoret uppfyllt, idx_lo och mid är båda 2, så idx_lo är inte mindre än mid längre. While-loopen avslutas.
Nu har vi tmp_arr=[2, 3, 4, 6, 0, 0, 0, 0].
På den sista raden används slicing för att kopiera värdena från tmp_arr index lo till n-1, dvs tmp_arr[0], tmp_arr[1], tmp_arr[2] och tmp_arr[3] till sort_arr[0], sort_arr[1], sort_arr[2] och sort_arr[3], så att tmp_arr = [2, 3, 4, 6, 1, 8, 5, 0] när funktionen avslutas.

b. Funktionen sätter ihop de två sorterade dellistorna från index lo till mid-1 och från mid till hi-1 i sort_arr till en sorterad lista, det vill säga vi sätter ihop [3, 6] och [2, 4] till [2, 3, 4, 6]. Det är merge-delen av mergesort.
c. Vi går högst igenom elementen från lo till hi-1 i funktionen, och tidskomplexiteten är linjär, O(N) eller O(hi-lo).
(Idx_lo börjar på lo och idx_hi börjar på mid. Vi börjar med några tilldelningar, konstant tidskomplexitet. I den första while-loopen har vi operationer med konstant tidskomplexitet i varje varv. While-loopen körs tills idx_lo har blivit mid eller tills idx_hi har blivit hi. Det betyder att det blir högst (hi-lo) varv eftersom ett av dem ökas på i varje varv, så den första while-loopen har linjär tidskomplexitet. I den andra while-loopen har vi också operationer med konstant tidskomplexitet. Den körs i högts mid-lo varv, eftersom idx_lo börjar på lo i början av funktionen och ökas på i varje varv. Det blir linjär tidskomplexitet (och varje varv som körs här motsvarar ett som aldrig gjordes i den första while-loopen för att idx_hi redan hade blivit hi). I raden där vi kopierar intervallet från lo till n-1 från tmp_arr till sort_arr har vi också linjär tidskomplexitet, eftersom det kan vara upp till hi-lo värden som flyttas. Vi har en del med konstant tidskomplexitet

följt av delar med linjär tidskomplexitet, den linjära växer snabbast och är resultatet.)

4. Det finns flera djupet-först-traverseringar som är möjliga och godkända. Ett exempel där vi väljer nästa nod i bokstavsordning är. 1. G, 2. C (från G), 3. B (från C), 4. A (från B), 5. F (från A), 6. D (från C).
   Ett exempel med omvänd bokstavsordning är: 1. G, 2. C (från G), 3. F (från C), 4. B (från F), 5. A (från B), . D (från C)
   Ett annat giltigt exempel är 1. G, 2. C (från G), 3. D (från C), 4. F (från C), 5. A (från F), 6. B (från A).

Lösningsförslag labbexamen, exempel

1. Se partition.py
2. Se SolBinarySearchTree.py