Implementarea heapsort versus selection sort ca un singur algoritm - cu două structuri de date diferite

Ciuperceanu Vlad-Mihai
Grupa 151

1 Aspecte ale proiectării codului

Cei doi algoritmi de sortare, $Heapsort$ și $Selection\,Sort$, au același principiu la bază, însă diferă implementarea, structura de date folosită și timpul final de execuție.

Principiul pe care funcționează cei doi algoritmi este de a porni dinspre finalul secvenței către începutul acesteia și de a găsi maximul din secvența neprocesată, pentru a-l aduce pe poziția curentă. Astfel, construim șirul sortat dinspre final înspre început, punând elementele în ordine descrescătoare.

De aceea, cele două funcții de sortare au codul aproape identic, tocmai pentru a evidenția similaritatea dintre ele, singura diferență fiind la apelarea funcției care găsește maximul și îl aduce pe poziția dorită, precum și la menținerea structurii de $Heap$.

Pentru $Selection\,Sort$, folosim funcția $findMaxAndSwap$, ce primește ca parametri lista de elemente și indicele de final al secvenței din care ne dorim să aflăm maximul. Aceasta face o simplă parcurgere pentru a găsi maximul, apoi îl interschimbă cu elementul de la poziția curentă. Din moment ce funcția parcurge toate elementele de la un indice dat către începutul șirului, aceasta va avea complexitatea $O(n)$, care, împreună cu parcurgerea inițială, face ca algoritmul de $Selection\,Sort$ să aibă complexitatea $O(n^2)$.

Pentru $Heapsort$, avem nevoie, mai întâi, să asigurăm structura de $Heap$, apelând, înainte de a începe sortarea, funcția $buildHeap$, ce se folosește de funcția $heapify$ pentru prima jumătate a elementelor date. Funcția $heapify$ ne ajută să menținem structura de $Heap$, în urma unor modificări făcute. Aceasta funcționează recursiv, aducând elementul maxim în nodul curent, urmând să verificăm proprietatea de $Heap$ și în fiii nodului. În funcția $heapSort$, codul este asemănător cu cel din $selectionSort$, cu două diferențe. Mai întâi, funcția $findMaxAndSwapHeap$ face același lucru ca funcția $findMaxAndSwap$, însă reușește acest lucru în $O(1)$, întrucât știm deja unde se află maximul, anume pe prima poziție, din structura de $Heap$. Apoi, pentru că am stricat structura heapului, vom apela funcția $heapify$ pentru a o menține. La fiecare pas, suntem

interesați să păstrăm avem un $Heap$ doar pentru elementele de la 1 la indicele curent, restul fiind deja pe pozițiile corecte, astfel că nu mai contează. De aceea, funcția $heapify$ are și un parametru pentru dimensiune, care se modifică odată cu indicele la care am ajuns. Cum găsirea și interschimbarea maximului se face în $O(1)$, iar $heapify$ are complexitatea $O(log\, n)$, algoritmul de $Heapsort$ va avea complexitatea $O(n\, log\, n)$.

2 Experimente

Pentru a testa diferențele de complexitate dintre cei doi algoritmi de sortare, am folosit biblioteca *numpy* pentru a genera permutări aleatoare cu primele 10000 de numere. Apoi am făcut câte o copie a listei și am contorizat cât timp au durat cele două sortări pentru aceeași secvență.

După câteva rulări, timpii obținuți au fost:

$Selection\, Sort$	$Heapsort$
1.981	0.047
1.932	0.050
2.033	0.049
2.224	0.046
1.985	0.047
1.887	0.074
1.938	0.061

Astfel că putem spune că $Heapsort$ este aproximativ cu 97% mai rapid decât $Selection\, Sort$, după cum reiese din testele de 10000 de elemente.