Alga 1

1. Mohó algoritmus

A mohó algoritmus alapelve, hogy minden lépésnél az optimális megoldást keresi. Ebből következik, hogy nem használhatjuk minden esetben viszont, ha hasznosnak minősül akkor nagyon jó eredményt ad. A részproblémákra bontásnál az a cél, hogy egyetlen részproblémát eredményezzen. Példa: hátizsák probléma.

Példa

A hátizsák problémánál a mohó algoritmus a következőképpen fog működni. Adott táska tárhelyéhez képest a legnagyobb értékű tárgyat fogja kiválasztani az algoritmus, amit súly alapján bele tud még tenni és rekurzívan mindig a lehető legnagyobb értékűt fogja választani mindaddig amíg a táska meg nem telik vagy nincs már olyan tárgy, ami beleférne. Ezáltal az algoritmus a rekurzív hívások számát a lehető legkisebbre csökkenti.

Ellenpélda

Globális maximum keresés hiszen itt a mohó algoritmus megkeresi az első inflexiós pontot (ahol előjelet vált) és ezt fogja eredményezni. Viszont ez lehet lokális maximum is, ami nem felel meg globális maximumnak.

2. Oszd meg és Uralkodj

diszjunkt: teljesen különálló

intuitív: önmagát megmagyarázó

Az oszd meg és uralkodj egy olyan probléma megoldó algoritmus, amely a problémát kisebb diszjunkt* részekre osztja (oszd meg) rekurzívan mindaddig amíg már nem lehet kisebb részekre osztani, ekkor megoldjuk a részproblémákat (uralkodás), majd ezeket a megoldott részproblémákat összevonjuk (összevonás) és ezt eredményezzük.

Példa

Adott egy függvény keressük rajta csúcsot

Vesszük a függvény felét hiszen, ha egy függvény felén találunk csúcsot akkor az a függvényben is csúcs lesz. Ezért vesszük a függvény felének a középpontját és megvizsgáljuk a tőle balra lévő értékeket, ha a középső érték a kisebb akkor folytatjuk a keresést a balra lévő értékekkel, ellenkező esetben pedig jobbra folytatjuk.

Minden oszd meg és uralkodj algoritmus megoldható iteratívan és rekurzívan is

Rekurzió előnyei:

Átláthatóbb a kód és intuitív*

Rekurzió hátrányai:

Memória allokáció és nehezebb debuggolni.

3. Dinamikus Programozás

A dinamikus programozás egy algoritmikai technika amely a problémát kisebb részproblémákra bontja ezeknek az eredményeit eltárolja és ezek felhasználásával oldja meg a nagy problémát. A módszer lényege, hogy az eltárolt megoldások segítségével nem kell a részproblémákat újra kiszámítani.

Pénzfelváltási példa adott N értékű pénzérmék és a legkevesebb érmét felhasználva, hogyan fizethető ki F forint?

Mohó algoritmus nem adna optimális megoldást

polinomális: O(n)

exponencionális: O(n2)

A nyers erő algoritmus, amely kiszámítja az összes lehetséges megoldást visszaadná az optimális megoldást viszont futásidőben exponencionális időkomplexitású, ezzel szemben a dinamikus programozás polinomális* időkomplexitású, mivel nem kell újra kiszámolni a már kiszámolt részproblémák eredményét a nyers erővel szemben. A részproblémák nem diszjunktak tehát átfedhetik egymást ezt kihasználva a dinamikus programozás felhasználja a már kiszámolt eredményeket ezáltal csökkenti a futásidőt (számítások számát).

4. Rendező algoritmusok

Beszúró rendezés: A beszúró rendezés egy egyszerű, de hatékony rendezési algoritmus kis adathalmazok esetén. (Olyan, mint amikor kártyát rendezünk a kezünkben: minden új kártyát a megfelelő helyre "illesztünk be" a már rendezett lapok közé.)

Így működik a beszúró rendezés:

1. Kezdjük a rendezést az adatsorozat második elemével, tekintve az első elemet már rendezettnek.
2. Vegyük ki a jelenlegi elemet, és hasonlítsuk össze az előző, már rendezett elemekkel.
3. Mozgassuk az összehasonlított rendezett elemeket egy pozícióval jobbra, amennyiben a jelenlegi elem kisebb, mint az összehasonlított elemek.
4. Ismételjük meg ezt a lépést addig, amíg meg nem találjuk a helyes pozíciót a jelenlegi elem számára a rendezett részen.
5. Helyezzük be a jelenlegi elemet a megfelelő helyre a rendezett részben.
6. Folytassuk ezt az eljárást a következő elemmel a sorban, amíg az összes elemet be nem szúrjuk a megfelelő helyre.

A beszúró rendezés előnyei közé tartozik az egyszerűsége, a stabil működés (nem változtatja meg azonos értékű elemek sorrendjét), valamint az, hogy helyben végezhető, tehát nem igényel plusz tárhelyet az adatok rendezéséhez. Azonban nagy adathalmazok esetén nem a leggyorsabb módszer, mivel az átlagos és a legrosszabb esetben is kvadratikus futási idővel (O(n^2)) rendelkezik, ahol 'n' az elemek száma.

Összefésülő: Az összefésülő rendezés vagy merge sort egy hatékony, összehasonlításon alapuló, általában növekvő sorrendben rendező algoritmus. A rendezési módszer a "megoszt és uralkodj" elven alapul, amely az alábbi lépésekből áll:

1. **Megosztás**: Az adatsorozatot rekurzívan két fele osztjuk, amíg már csak egyelemű (vagy üres) részlistákat nem kapunk, amelyek már rendezettnek tekinthetők.
2. **Összefésülés**: Az egyelemű listákat kétessével összefésüljük úgy, hogy az összefésült lista rendezett legyen. Ez azt jelenti, hogy a két lista elemeit összehasonlítjuk, és a kisebb elemet választjuk ki elsőként az összefésült listába, majd így haladunk tovább, amíg az összes elemet be nem soroljuk.
3. **Ismétlés**: Az összefésülést addig ismételjük, amíg az eredeti lista elemeit nem rendezzük teljesen egyetlen listába.

A merge sort algoritmus egy példával szemléltetve:

Tegyük fel, hogy rendezni szeretnénk a következő egész számok sorozatát: [6, 5, 3, 1, 8, 7, 2, 4]

1. **Megosztás**:
    - [6, 5, 3, 1] és [8, 7, 2, 4]
2. Tovább bontjuk őket, amíg egyelemű részlisták nem lesznek:
    - [6, 5], [3, 1], [8, 7], [2, 4]
    - [6], [5], [3], [1], [8], [7], [2], [4]
3. **Összefésülés**:
    - [5, 6] és [1, 3], valamint [7, 8] és [2, 4]
4. Az összefésülést folytatjuk:
    - [1, 3, 5, 6] és [2, 4, 7, 8]
5. Végül egy nagy összefésüléssel rendezett listát kapunk:
    - [1, 2, 3, 4, 5, 6, 7, 8]

Az összefésülő rendezés hatékony az adatok rendezésében, és a legrosszabb, átlagos és legjobb esetben is $O(n*\log n)$ időkomplexitással rendelkezik, ahol 'n' az elemek számát jelöli. Ez a hatékonyság abból adódik, hogy minden osztási

lépésben az elemek száma feleződik, és az összefésülés minden szintjén az elemek számának arányában nő a lépések száma. Stabil algoritmus, tehát az azonos értékű elemek relatív sorrendje nem változik meg a rendezés során.

Kupac: A kupacrendezés (heap sort) egy hatékony rendezési algoritmus, amely a kupac adatszerkezetet használja az elemek rendezéséhez. A kupac olyan bináris fa, ahol minden szülőcsomópont értéke nagyobb (vagy kisebb, attól függően, hogy maximum vagy minimum kupacot használunk) mint a gyermekcsomópontjaié, így a fa gyökerében mindig a legnagyobb (vagy legkisebb) elem található.

A kupacrendezés fő lépései:

1. **Kupac létrehozása**: Az összes elemet tartalmazó tömböt átalakítjuk kupaccá. Ezt úgy érjük el, hogy az utolsó szülőcsomóponttól kezdve minden csomópontot "heapify" (kupacol) művelettel rendezünk a gyerekeihez képest. Ez biztosítja, hogy minden szülő nagyobb értékű legyen mint a gyerekei.
2. **Rendezés**:
    - Az algoritmus eltávolítja a legnagyobb elemet (a kupac gyökerét), amelyet a tömb végére helyez.
    - A tömb hátralévő, még rendezetlen részén újra "heapify" műveletet hajtunk végre, helyreállítva a kupac tulajdonságait.
    - Ismételjük ezt a lépést, amíg az összes elem meg nem cserélődik, és a tömb végére nem kerülnek rendezetten.
3. **Heapify (Kupacol)**:
    - A heapify művelet egy adott csomópontban ellenőrzi, hogy a szülő nagyobb-e, mint a gyerekei. Ha nem, akkor csere történik a nagyobb gyermekkel, és a heapify műveletet rekurzívan megismételjük a lecserélt gyermekre.

A kupacrendezés azért hatékony, mert a kupac legfelső elemének eltávolítása után gyorsan újra tudjuk rendezni a kupacot a heapify művelettel. A rendezés időkomplexitása általában $O(n \log n)$, ahol 'n' az elemek számát jelöli, mivel minden elem esetében a heapify művelet legrosszabb esetben is logaritmikus ideig tart (hiszen egy kupac magassága logaritmikus a csomópontok számához képest).

Összefoglalva a kupacrendezés egy nagyon hatékony rendezési algoritmus, különösen nagy adathalmazok esetén, és helyben (in-place) végrehajtható, ami

azt jelenti, hogy nem igényel plusz tárhelyet az elemek rendezéséhez (kivéve a rekurzív heapify művelet stack memóriáját).

Gyorsrendezés: A gyorsrendezés vagy quicksort egy hatékony, általános célú, összehasonlításon alapuló rendezési algoritmus. Az "oszd meg és uralkodj" elven működik, és a következő lépésekből áll:

1. **Pivot választás**: Először válasszunk ki egy "pivot" elemet az adathalmazból. A pivot lehet bármelyik elem, például az első, az utolsó, a középső, vagy akár egy véletlenszerűen választott elem. A pivot feladata az, hogy segítsen az elemek felosztásában.
2. **Partícionálás**: Ezután rendezzük át a tömböt úgy, hogy a pivotnál kisebb elemek kerüljenek a pivot bal oldalára, a nagyobbak pedig a jobb oldalára. A partícionálás után a pivot elem a végső helyére kerül, és minden a pivotnál kisebb elem a bal oldalra, minden nagyobb elem pedig a jobb oldalra kerül.
3. **Rekurzív megosztás**: A tömböt ezen a ponton két részre osztottuk: a pivot elem előtti részre és a pivot elem utáni részre. Ezt a folyamatot rekurzívan alkalmazzuk a pivot elem bal és jobb oldalán lévő részekre, amíg a részek mérete nem csökken egyeleműé vagy üressé.
4. **Összefésülés**: Mivel a rekurzív hívások nem igényelnek további összefésülést (a quicksort helyben végezhető el), a tömb rendezetté válik, amint minden rekurzív hívás befejeződik.

A gyorsrendezés hatékonyságának kulcsa a hatékony partícionálásban és a megfelelő pivot választásban rejlik. Rossz pivot-választás esetén az algoritmus futási ideje lelassulhat és $O(n2)$-re nőhet, de jó pivot-választás esetén általában $O(n\log n)$ időkomplexitással rendelkezik.

Bár a gyorsrendezés átlagosan gyors és hatékony, nem stabil rendezés, ami azt jelenti, hogy az azonos értékű elemek eredeti sorrendje megváltozhat. Emellett rossz pivot-választás (például ha a tömb már rendezett és az első vagy utolsó elemet választjuk pivotnak) esetén a teljesítménye jelentősen csökkenhet.

Leszámoló rendezés: A leszámláló rendezés (counting sort) egy nem összehasonlításon alapuló rendezési algoritmus, amely különösen akkor hatékony, amikor a rendezendő értékek kis tartományba esnek. A leszámláló rendezés nem az elemek egyedi összehasonlítását használja, hanem az elemek értékeinek gyakoriságát számolja meg. Íme, hogyan működik a leszámláló rendezés:

1. **Gyakorisági tömb létrehozása**: Először készítsünk egy gyakorisági tömböt (vagy számláló tömböt), amely minden lehetséges értékhez tartozóan tárolja, hogy hányszor fordul elő az adott érték az eredeti tömbben. Például, ha a legkisebb érték 0 és a legnagyobb érték k, akkor létrehozunk egy (k+1)-hosszúságú gyakorisági tömböt.
2. **Gyakoriságok számolása**: Menjünk végig az eredeti tömbön, és minden elem esetében növeljük az adott értéknek megfelelő indexű gyakorisági tömb elemét.
3. **Gyakorisági tömb kumulált értékeinek számítása**: Módosítsuk a gyakorisági tömböt úgy, hogy minden elem az adott értékig összes előfordulását tartalmazza. Ez azt mutatja majd, hogy az adott értékű vagy kisebb elemekből összesen hány darab van.
4. **Eredmény tömb létrehozása**: Hozzunk létre egy eredmény tömböt, amely ugyanolyan hosszúságú, mint az eredeti tömb.
5. **Elemek besorolása az eredmény tömbbe**: Menjünk végig az eredeti tömbön a végétől kezdve (ez biztosítja a stabilitást, azaz az azonos értékű elemek eredeti sorrendje megmarad), és használjuk a gyakorisági tömböt az elemek helyének meghatározásához az eredmény tömbben. Ahogy egy elemet besorolunk, csökkentsük a megfelelő értékű gyakorisági tömb elemét.
6. **Másolás az eredeti tömbbe**: Másoljuk át az eredmény tömb tartalmát az eredeti tömbbe.

A leszámláló rendezés hatékony, amikor a rendezendő elemek tartománya kicsi a tömb méretéhez képest, mivel ilyenkor a gyakorisági tömb nem lesz túl nagy, és az algoritmus $O(n+k)$ időben képes futni, ahol n az elemek száma, k pedig a lehetséges értékek tartománya. Stabil algoritmus, tehát azonos értékű elemek eredeti sorrendje nem változik meg a rendezés során.

5. Gráfalgoritmusok

Irányítatlan

Minimális feszítőfa probléma egy olyan irányítatlan gráf, amely G=(V,E) ahol V a csúcsokét az E pedig az élek halmazát jelzi. Szerepel még egy gráf is a feladatban ami pedig a élsúlyok költségét jelölik. A feladat pedig egy feszítőfa létrehozása amely minden csúcsot érint és az élek költsége minimális legyen.

Kruskal: Ennek a módszernek a lényege, hogy megkeressük a legkisebb költségű élt és behúzzuk, utána megkeressük a következő legkisebb költségű élt és azt is behúzzuk és így haladunk. Nem szükséges, hogy a csúcsok folyamatosan fát alkossanak viszont arra ügyelni kell, hogy ne keletkezzen kör a gráfban. Az algoritmus akkor ér véget amikor minden csúcshoz tartozik él és egy körmentes gráfot kapunk. Az algoritmus futásideje logaritmikus O(E*logV). Mohó

Prim: Ennek a módszernek a lényege, hogy megkeressük a legkisebb költségű élt, azt behúzzuk majd az összekötött csúcsokhoz tartozó élek minimum költségét keressük, ami még nem a fa része és behúzzuk. Az algoritmus futásideje logaritmikus O(E*logV). Mohó

Irányított

Minimális feszítőfa probléma egy irányított gráfon a cél, hogy minden csúcs elérhető legyen az élek által egy kiinduló csúcsból.

Djikstra: A Dijkstra algoritmus egy gráf minden csúcsához társít egy költséget, amely kezdetben végtelen (a kezdő csúcs költsége 0). Az algoritmus minden iterációjában kiválasztja azt a még nem látogatott csúcsot, amelyikhez a legkisebb költségen keresztül lehet eljutni, frissíti a szomszédos csúcsokhoz vezető költségeket, ha az adott csúcson keresztül olcsóbban elérhetőek, majd az adott csúcsot "látogatottnak" jelöli. Az algoritmus folytatódik, amíg minden csúcsot meg nem látogattunk vagy amíg a célcsúcs költségét meg nem határoztuk. Futásideje logaritmikus O(E*logV). Mohó

Negatív élek esetén nem lehet Djikstrát használni -> Bellman-Ford Futásidő (V*E)

Szélességi keresés

A szélességi bejárás (szélességi keresés) egy gráfban történő keresési módszer, amely a gráf csúcsait szintek szerint, azaz a kezdőcsúcstól való távolságuk alapján vizsgálja meg. Az algoritmus célja, hogy megtalálja az összes csúcsot a gráfban a kiindulási ponttól kezdve, a legközelebbi csúcsoktól az egyre távolabbiakig haladva.

A keresési folyamat egy kiválasztott csúccsal kezdődik, amit gyakran "kezdőcsúcsnak" nevezünk. Ezt a csúcsot megjelöljük, mint "felfedezett", és a keresési sorba helyezzük.

A keresési sor a meglátogatandó csúcsok listáját tartalmazza. Kezdetben csak a kezdőcsúcs van benne.

Amíg a sor nem üres, kivesszük a sor elején lévő csúcsot (ez a jelenlegi csúcs), és megvizsgáljuk a szomszédait. Minden még nem látogatott szomszédot "felfedezettnek" jelölünk, és hozzáadjuk a sor végéhez. Ezzel egyúttal feljegyezzük az adott szomszéd távolságát a kezdőcsúcstól, ami mindig egyel több, mint a jelenlegi csúcs távolsága.

Miután egy csúcs összes szomszédját felfedeztük és a sorhoz adtuk, a csúcsot "teljesen vizsgáltnak" jelöljük, ami azt jelenti, hogy már nem kell többé foglalkoznunk vele.

Ezt az eljárást ismételjük minden sorban lévő csúcsra, amíg a sor ki nem ürül, vagyis amíg az összes elérhető csúcsot meg nem látogatjuk.

A szélességi bejárás eredményeképpen képesek vagyunk meghatározni a legrövidebb utat a kezdőcsúcstól minden elérhető csúcsig a gráfban, mivel mindig a lehető legközelebbi, még nem vizsgált csúcsot vizsgáljuk meg először.

Mélységi keresés

A mélységi keresés egy gráfbejáró algoritmus, amely a gráf csúcsait a lehetséges legmélyebb csúcsig követi az éleket, mielőtt visszatérne és más útvonalat is kipróbálna.

A keresés egy kiválasztott kezdőcsúcsból indul. Ezt a csúcsot "felfedezettnek" jelöljük, amint elkezdjük a keresést.

A kezdőcsúcstól kiindulva, a DFS az egyik szomszédos csúcsra lép, amely még nem volt felfedezve, és ezt a csúcsot is "felfedezettnek" jelöljük.

A DFS tovább mélyed a gráfban, azaz minden lépésben a jelenlegi csúcs egy még nem látogatott szomszédjára lép tovább, és ezt az új csúcsot is "felfedezettnek" jelöljük. Ez folytatódik addig, amíg el nem érünk egy olyan csúcshoz, amelynek nincsenek látogatatlan szomszédai, vagy egy "záró" csúcshoz, amelyből nem lehet tovább haladni.

Ha egy csúcsnál nem tudunk tovább haladni, "visszalépünk" az előző csúcshoz (ezt hívjuk backtrackingnek), és onnan keresünk új, még nem látogatott szomszédokat.

Ez a folyamat addig ismétlődik, amíg minden elérhető csúcsot meg nem látogattunk a gráfban, vagy amíg nem érjük el a keresés célját.

A mélységi keresés során minden csúcsot, amit már felfedeztünk és ahonnan további lépéseket tervezünk, egy verem (stack) segítségével jegyezhetünk fel. Amikor "visszalépünk", a veremből vesszük ki az előző csúcsot, amelyhez vissza kell térnünk. Ez a verem segítségével történő backtracking biztosítja, hogy minden lehetséges utat végig tudjunk követni a gráfban.

A mélységi keresés különösen hasznos olyan problémák megoldásában, ahol fontos minden lehetséges útvonal kipróbálása, például labirintusokban való út keresésekor vagy gráfokban való ciklusok felfedezésekor.

Futásidők:

Mohó: n*logn

Oszd meg és uralkodj: n*logn

Dinamikus prog: n

Beszúró rendezés: n*n

Összefésülő: n*logn

Kupac: n*logn

Gyorsrendezés: n*logn (legrosszabb n*n)

Leszámoló: n+k

Kruskal: E*logV (mohó)

Prím: E*logV (mohó)

Djikstra: E*logV (Mohó)

Szélességi bejárás: V+E

Mélységi bejárás: V+E