Algoritmusok bonyolultsága; Bináris keresés és alkalmazásai

Felvételi felkészítő

Dr. Păţcaş Csaba

Babeş-Bolyai Tudományegyetem
Magyar Matematika és Informatika Intézet

2024.02.23

Bevezető

- Bemutatkozás
- Online kommunikáció: mindenki lenémítva, kézfelemelés
- 3×50 perc
- Kérdések

Milyen szempontból lehet „hatékony” egy algoritmus?

Milyen szempontból lehet „hatékony” egy algoritmus?

- Futási idő
- Memóriaigény
- Tárhelyigény
- Klasszikus algoritmusok esetén nem szoktuk említeni, de párhuzamosított algoritmusok esetén ide tartozhat a kommunikációhoz szükséges sávszélesség is (lásd pl. bitcoin bányászat). Ez fontos szempont lehet bármilyen esetben ahol adatokat küldünk interneten keresztül, például webes alkalmazások esetén, kliens-szerver architektúrákban stb.

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

Mi a probléma a direkt időméréssel?

Hogyan hasonlíthatjuk össze két algoritmus hatékonyságát?

Mi a probléma a direkt időméréssel?

- Függ a számítógép(hálózat) hardware-jétől.
- Függ a használt programozási nyelvtől.
- Függ a fordítóprogram verziójától.
- Függ a használt optimalizálási szinttől.
- Függ az implementálás módjától (pl. paraméterátadás).
- Függ az operációs rendszertől.
- Függ a bemeneti adatok méretétől és szerkezetétől.

Legrosszabb, legjobb, átlagos esetek

Feladat

Adott egy n elemű a tömb és egy x érték. Állapítsuk meg, hogy x szerepel-e a tömbben!

- Válasszuk alapműveletnek az összehasonlítást!
- A legjobb esetben rögtön az első pozición megtaláljuk a keresett elemet, ekkor a műveletek száma 1.
- A legrosszabb esetben x nem szerepel a tömbben, de ezt csak n összehasonlítás után tudjuk megállapítani.
- Kis matekezéssel könnyedén levezethető, hogy az átlagos esetben $\frac{n+1}{2}$ összehasonlításra van szükségünk.

Algoritmusok növekedési rendje

- A fentiekből lekövetkeztethetjük, hogy nem érdemes, de nem is lehetséges pontos, precíz értékeket keresni a futási idő megadására (ez sok esetben a többi hatékonysági szempontra is igaz).
- Az esetek túlnyomó többségében arra kell szorítkoznunk, hogy a végrehajtási idő nagyságrendjét határozzuk meg.
- Láttuk, hogy ezt kézenfekvő a bemeneti adatok méretének függvényében kifejezni, ezt általában n-el jelöljük (vannak esetek, amikor nem elég egy paraméter a bemeneti adatok méretének kifejezéséhez).

- Az előbbi példában egy vektor volt a bemeneti adat, ilyenkor n a tömb elemeinek számát jelöli.
- Két mátrix összeszorzásakor a mátrixok sorainak és oszlopainak számát tekintjük a bemeneti adatok méretének.
- Nagy számok összeadásakor a számok számjegyeinek száma a méret.
- Gráfelméleti feladatokban általában a csomópontok számát (n) és sokszor az élek számát (m) is figyelembe vesszük.

Algoritmusok növekedési rendje

Az alapművelet

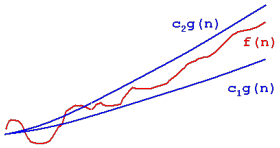
- Amikor egy algoritmus hatékonyságát próbáljuk meghatározni futási idő szempontjából, annak a nagyságrendjét próbáljuk kifejezni, hogy egy adott alapművelet hányszor hajtódik végre.
- Fontos, hogy hogyan választjuk meg ezt a műveletet, sőt előfordulhat, hogy több alapműveletet is kell választanunk a pontos eredmény érdekében.
- Jellemzően a „legbelsőbb” műveletekre kell gondolnunk, amelyek a legtöbbször hajtódnak végre.

Algoritmusok növekedési rendje

- Az algoritmusok bonyolultságát (komplexitását) a bemeneti adatok méretének (n, m stb.) függvényeként írjuk le.
- A képletnek csak a fő tagját tartjuk meg, pl. $an^2 + bn + c$-ből csak az an^2-et, mivel az alacsonyabb rendű tagok nagy n-re kevésbé lényegesek.
- Szintén figyelmen kívül hagyjuk a fő tag konstans szorzóját, mivel nagy bemenetekre ez is elhanyagolható, így csak n^2 marad a fenti példában.
- Az így kapott kifejezést nevezzük az algoritmus növekedési rendjének.
- Általában akkor mondjuk, hogy egy algoritmus hatékonyabb egy másiknál, ha a legrosszabb esetben való növekedési rendje kisebb.

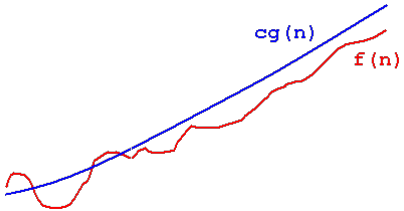
- Formálisan
  $\Theta(g(n)) = \{f(n) | \exists c_1, c_2, n_0 > 0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$
- Az aszimptotikus jelölések függvények halmazát jelölik, vagyis matematikailag úgy lenne pontos írni, hogy $3n^2 + 5n + 2 \in \Theta(n^2)$, de a jelöléseket kicsit feloldva $f(n) = \Theta(g(n))$-t szoktuk használni.
- A Θ jelölést szavakkal úgy is leírhatjuk, hogy minden $n \geq n_0$ esetén az $f(n)$ függvény – egy állandó szorzótényezőtől eltekintve – egyenlő $g(n)$-el.
- Ezt úgy mondjuk, hogy $g(n)$ aszimptotikusan éles korlátja $f(n)$-nek.

- Más szóval az $f(n)$ függvény hozzátartozik a $\Theta(g(n))$ halmazhoz, ha léteznek c_1 és c_2 állandók úgy, hogy $f(n)$ – elég nagy n-re – beszorítható $c_1 g(n)$ és $c_2 g(n)$ közé.
- Grafikusan



A konstans (állandó) bonyolultságot $\Theta(1)$-el jelöljük, ekkor az algoritmus futási ideje nem függ a bemeneti adatok méretétől, vagy ha memóriáról beszélünk nem használ n-el arányos méretű plusz memóriát.

- Az O jelölés aszimptotikus felső korlátot ad.
- $O(g(n)) = \{f(n) | \exists c, n_0 > 0 : 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$



Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt?

Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10^8$
- Mekkora n-re fut le egy $\Theta(n^2)$ időbonyolultságú algoritmus pár másodperc alatt?

Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10^8$
- Mekkora n-re fut le egy $\Theta(n^2)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10000$
- Mekkora n-re fut le egy $\Theta(n^3)$ időbonyolultságú algoritmus pár másodperc alatt?

Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10^8$
- Mekkora n-re fut le egy $\Theta(n^2)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10000$
- Mekkora n-re fut le egy $\Theta(n^3)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 500$
- Mekkora n-re fut le egy $\Theta(n \log n)$ időbonyolultságú algoritmus pár másodperc alatt?

Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10^8$
- Mekkora n-re fut le egy $\Theta(n^2)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10000$
- Mekkora n-re fut le egy $\Theta(n^3)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 500$
- Mekkora n-re fut le egy $\Theta(n \log n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 5\ 000\ 000$
- Mekkora n-re fut le egy $\Theta(\log n)$ időbonyolultságú algoritmus pár másodperc alatt?

Mindez a gyakorlatban 2024-ben

- Mekkora n-re fut le egy $\Theta(n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10^8$
- Mekkora n-re fut le egy $\Theta(n^2)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 10000$
- Mekkora n-re fut le egy $\Theta(n^3)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 500$
- Mekkora n-re fut le egy $\Theta(n \log n)$ időbonyolultságú algoritmus pár másodperc alatt? kb. $n \leq 5\ 000\ 000$
- Mekkora n-re fut le egy $\Theta(\log n)$ időbonyolultságú algoritmus pár másodperc alatt? Gyakorlatilag bármekkorára :)

Az algoritmus által feldolgozott adatok számára szükséges memória mérete

- Gyakran előfordul, hogy egy program a bemeneti és kimeneti adatokon kívül, ideiglenesen létrehozott adatszerkezetekkel is dolgozik.
- Amikor egy algoritmus memóriabonyolultságáról beszélünk, akkor általában ezekre gondolunk, tehát a bemeneti adatok tárolására szükséges memóriát nem vesszük figyelembe.
- Például ha egy segédmátrixot használna az algoritmusunk, a memóriabonyolultság $\Theta(n^2)$ lenne.
- Ha egy algoritmus feldolgozás közben csak konstans méretű plusz memóriát vesz igénybe (vagyis memóriabonyolultsága $\Theta(1)$), azt mondjuk, hogy helyben dolgozik. Ekkor a lefoglalt memória mérete nem függ a bemeneti adatok méretétől.

Bonyolultsági osztályok

- Az informatikában megjelenő feladatokat különböző bonyolultsági osztályokba sorolhatjuk, ezek közül számunkra a legfontosabbak a `P`, az `NP` és az `NP-teljes` osztályok.
- A `P` vagy `PTIME` bonyolultsági osztályba azok a feladatok tartoznak, melyek megoldhatóak polinomiális időben, vagyis a legrosszabb esetben vett időbonyolultságuk $n^{O(1)}$ formában írható. Például: osztja-e egyik szám a másikat; annak ellenőrzése, hogy egy szám prím-e
- A matematikai precizitást elhagyva, köznyelvileg mondhatjuk, hogy az `NP` bonyolultsági osztályba azok a feladatok tartoznak, melyek eredménye ellenőrízhető polinomiális időben. Például: a Hamilton-kör létezésének kérdésében a csúcsok egy sorozatáról könnyen ellenőrízhető, hogy egy Hamilton-kört írnak-e le; ha a kérdés az, hogy ki tudunk-e pontosan fizetni egy adott összeget bizonyos érmékkel, a kiválasztott érmék értékeit csak össze kell adnunk az ellenőrzéshez.

P és NP kapcsolata

P = NP vagy P ≠ NP?

Könnyen belátható, hogy $\mathtt{P} \subseteq \mathtt{NP}$, de az máig nyitott kérdés, hogy a két feladatosztály egyenlő-e, vagy $\mathtt{P} \subset \mathtt{NP}$. A szakértők túlnyomó többsége szerint a második eset áll fenn, de ezt még nem sikerült bizonyítani.

NP-teljes feladatok

- Az NP-teljes feladatok a legnehezebb (legáltalánosabb) feladatok az NP osztályból.
- Bármelyik NP feladat levezethető (redukálható) bármely NP-teljes feladatra.
- Ebből következik, hogy az NP-teljes feladatok egymásra redukálhatóak.

Könnyű és nehéz feladatok

- Általában azokat a feladatokat tekintjuk könnyen megoldhatónak (tractable), amelyek megoldására ismerünk polinomiális idejű algoritmust, vagyis a P feladatosztályba tartoznak.
- Ezekkel ellentétben vannak a nehezen megoldható (intractable) feladatok, melyeknek legfontosabb csoportját az NP-teljes feladatok képezik.
- Az NP-teljes feladatokat azért fontos ismerni, mert ha egy feladatról tudjuk, hogy nehezen megoldható, abból következik, hogy nagy eséllyel nincs értelme polinomiális megoldási algoritmust keresnünk rá.
- Ismeretlen feladatokról is gyakran úgy bizonyítjuk, hogy nehezen megoldhatóak, hogy levezetjük őket egy ismert NP-teljes feladatra.

Mihez kezdhetünk a nehéz feladatokkal?

- Kis bemenetekre alkalmazhatunk nyers erőre, vagy visszalépéses keresésre alapuló megközelítéseket.
- Ha egy gyorsan előállítható megoldásra van szükségünk, valamilyen greedy heurisztikán alapuló stratégia jó választás lehet.
- Bizonyos feladatokra léteznek közelítő algoritmusok, amelyek garantált hibaszázalékon belül maradnak az optimumhoz képest.
- Valamivel nagyobb futási időt igényelnek, de sokszor lényegesen jobb megoldásokkal szolgálnak, a mesterséges intelligencia területéről ismert különböző metaheurisztikák. Ezek egy része nemdeterminisztikus algoritmus, vagyis véletlenszámokra alapszik, így két egymás utáni futás nem mindig ad azonos eredményt.

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = 1, n végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz:

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = 1, n végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz: $\Theta(n^2)$

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i, n végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz:

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i, n végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz: $\Theta(n^2)$

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i, n, i végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz:

Mennyi az alábbi algoritmus időbonyolultsága?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i, n, i végezd el:
    Kiír: '*'
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz: $\frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{n} = n \cdot (\frac{1}{2} + \ldots + \frac{1}{n}) = \Theta(n \log n)$

Mennyi az alábbi algoritmus időbonyolultsága?

```
i = n
AMÍG i > 1 végezd el:
  MINDEN j = 1, i végezd el:
    Kiír: '*'
  VÉGE(Minden)
  i = i / 2
VÉGE(Amíg)
```

Helyes válasz:

Mennyi az alábbi algoritmus időbonyolultsága?

```
i = n
AMÍG i > 1 végezd el:
  MINDEN j = 1, i végezd el:
    Kiír: '*'
  VÉGE(Minden)
  i = i / 2
VÉGE(Amíg)
```

Helyes válasz: $\frac{n}{2^0} + \frac{n}{2^1} + \ldots + \frac{n}{2^k} = n \cdot (\frac{1}{2^0} + \ldots + \frac{1}{2^k}) = \Theta(n)$, ahol $k \simeq \lfloor \log_2 n \rfloor$

Legyen a következő algoritmus, amelynek paramétere az n, nullától különböző természetes szám és amely egy természetes számot térít vissza.

```
Algoritmus f(n):
   j ← n
   Amíg j > 1 végezd el
      i ← 1
      Amíg i ≤ n végezd el
         i ← 2 * i
      vége(amíg)
      j ← j DIV 3
   vége(amíg)
   visszatérít j
Vége(algoritmus)
```

A következő bonyolultsági osztályok közül melyikhez tartozik hozzá a fenti algoritmus időbonyolultsága?

a. $O(\log_2 n)$
b. $O(\log_2^2 n)$
c. $O(\log_3^2 n)$
d. $O(\log_2 \log_3 n)$

Helyes válasz:

Legyen a következő algoritmus, amelynek paramétere az n, nullától különböző természetes szám és amely egy természetes számot térít vissza.

```
Algoritmus f(n):
   j ← n
   Amíg j > 1 végezd el
      i ← 1
      Amíg i ≤ n végezd el
         i ← 2 * i
      vége(amíg)
      j ← j DIV 3
   vége(amíg)
   visszatérít j
Vége(algoritmus)
```

A következő bonyolultsági osztályok közül melyikhez tartozik hozzá a fenti algoritmus időbonyolultsága?

a. $O(\log_2 n)$
b. $O(\log_2^2 n)$
c. $O(\log_3^2 n)$
d. $O(\log_2 \log_3 n)$

Helyes válasz: b, c

Az alábbi algoritmusok közül melyek implementálhatók úgy, hogy időbonyolultságuk lineáris legyen ($O(n)$)?

a) Egy elem szekvenciális keresése egy n számot tartalmazó vektorban
b) Egy n számot tartalmazó egydimenziós tömb rendezése beszúró rendezéssel
c) A legnagyobb érték megkeresése egy n számot tartalmazó nem rendezett vektorban
d) Egy n soros és n oszlopos négyzetes mátrix főátlóján található elemek összegének kiszámítása

Helyes válasz:

Az alábbi algoritmusok közül melyek implementálhatók úgy, hogy időbonyolultságuk lineáris legyen ($O(n)$)?

a. Egy elem szekvenciális keresése egy n számot tartalmazó vektorban
b. Egy n számot tartalmazó egydimenziós tömb rendezése beszúró rendezéssel
c. A legnagyobb érték megkeresése egy n számot tartalmazó nem rendezett vektorban
d. Egy n soros és n oszlopos négyzetes mátrix főátlóján található elemek összegének kiszámítása

Helyes válasz: a, c, d

Adott az `f(x, n)` algoritmus, ahol x és n természetes számok $(0 < n \leq 10000, 0 < x \leq 10000)$

```
1. Algorithm f(x, n):
2.     If n = 0 then
3.         return 1
4.     EndIf
5.     m ← n DIV 2
6.     p ← f(x, m)
7.     If n MOD 2 = 0 then
8.         return p * p
9.     EndIf
10.    return x * p * p
11.EndAlgorithm
```

Feltételezve, hogy minden szorzási és osztási művelet végrehajtásának ideje konstans, mit mondhatunk az algoritmus időbonyolultságáról?

a) Az időbonyolultság függ az x és n paraméterektől.

b) Az időbonyolultság nem függ az x paramétertől.

c) Az időbonyolultság $O(\log \log n)$.

d) Az időbonyolultság logaritmikus az n paraméter függvényében ($O(logn)$).

Helyes válasz:

Adott az `f(x, n)` algoritmus, ahol x és n természetes számok $(0 < n \leq 10000, 0 < x \leq 10000)$

```
1. Algorithm f(x, n):
2.     If n = 0 then
3.         return 1
4.     EndIf
5.     m ← n DIV 2
6.     p ← f(x, m)
7.     If n MOD 2 = 0 then
8.         return p * p
9.     EndIf
10.    return x * p * p
11.EndAlgorithm
```

Feltételezve, hogy minden szorzási és osztási művelet végrehajtásának ideje konstans, mit mondhatunk az algoritmus időbonyolultságáról?

a) Az időbonyolultság függ az x és n paraméterektől.

b) Az időbonyolultság nem függ az x paramétertől.

c) Az időbonyolultság $O(\log \log n)$.

d) Az időbonyolultság logaritmikus az n paraméter függvényében ($O(logn)$).

Helyes válasz: b, d

Adott az alábbi kódrészlet. Adjátok meg, hogy hányszor lesz kiírva az 'UBB' karakterlánc, tudva, hogy $n = 3^k$, ahol k természetes szám $(1 \leq k \leq 30)$?

```
j ← n
While j > 1 execute
    i ← 1
    While i ≤ n execute
        i ← 3 * i
        Write 'UBB'
    EndWhile
    j ← j DIV 3
EndWhile
```

a) k^2

b) $k \cdot 3^k$

c) $k(k+1)$

d) $3k$

Helyes válasz:

Adott az alábbi kódrészlet. Adjátok meg, hogy hányszor lesz kiírva az 'UBB' karakterlánc, tudva, hogy $n = 3^k$, ahol k természetes szám ($1 \leq k \leq 30$)?

```
j ← n
While j > 1 execute
    i ← 1
    While i ≤ n execute
        i ← 3 * i
        Write 'UBB'
    EndWhile
    j ← j DIV 3
EndWhile
```

a) k^2

b) $k \cdot 3^k$

c) $k(k+1)$

d) $3k$

Helyes válasz: c

Melyik bonyolultsági osztály írja le legpontosabban az alábbi algoritmus időbonyolultságát?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i + 1, n végezd el:
    segéd = n
    AMÍG (segéd > 3) végezd el:
      segéd = segéd / 4
    VÉGE(Amíg)
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz:

Melyik bonyolultsági osztály írja le legpontosabban az alábbi algoritmus időbonyolultságát?

```
MINDEN i = 1, n végezd el:
  MINDEN j = i + 1, n végezd el:
    segéd = n
    AMÍG (segéd > 3) végezd el:
      segéd = segéd / 4
    VÉGE(Amíg)
  VÉGE(Minden)
VÉGE(Minden)
```

Helyes válasz: $\Theta(n^2 \log n)$

Mennyi a beszúró rendezés memóriabonyolultsága a legrosszabb esetben?

a. $\Theta(1)$
b. $\Theta(\log n)$
c. $\Theta(n)$
d. $\Theta(n \log n)$
e. $\Theta(n^2)$

Helyes válasz:

Mennyi a beszúró rendezés memóriabonyolultsága a legrosszabb esetben?

a. $\Theta(1)$
b. $\Theta(\log n)$
c. $\Theta(n)$
d. $\Theta(n \log n)$
e. $\Theta(n^2)$

Helyes válasz: a

Az alábbiak közül melyik bonyolultsági osztályokhoz tartozik a bináris keresés időbonyolultsága legrosszabb esetben?

a. $\Theta(\log n)$
b. $\Theta(n)$
c. $\Theta(n \log n)$
d. $O(\log n)$
e. $O(n)$
f. $O(n \log n)$

Helyes válasz:

Az alábbiak közül melyik bonyolultsági osztályokhoz tartozik a bináris keresés időbonyolultsága legrosszabb esetben?

a. $\Theta(\log n)$
b. $\Theta(n)$
c. $\Theta(n \log n)$
d. $O(\log n)$
e. $O(n)$
f. $O(n \log n)$

Helyes válasz: a, d, e, f

Mennyivel nő a `számláló` változó értéke miután az alábbi alprogramot `P(1)` alakban hívjuk meg? Feltételezhetjük, hogy az algoritmus helyesen lefut és n páros szám.

```
ALGORITMUS P(i)
  HA (i > n) akkor
    számláló = számláló + n
  KÜLÖNBEN
    P(i + 2)
    P(i + 2)
    P(i + 2)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Helyes válasz:

Mennyivel nő a `számláló` változó értéke miután az alábbi alprogramot `P(1)` alakban hívjuk meg? Feltételezhetjük, hogy az algoritmus helyesen lefut és n páros szám.

```
ALGORITMUS P(i)
  HA (i > n) akkor
    számláló = számláló + n
  KÜLÖNBEN
    P(i + 2)
    P(i + 2)
    P(i + 2)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Helyes válasz: $n \cdot 3^{\frac{n}{2}}$

Legyen a következő algoritmus, amelynek paramétere az n, nullától különböző természetes szám és amely egy természetes számot térít vissza.

```
Algoritmus f(n):
  j ← n
  Amíg j > 1 végezd el
    i ← 1
    Amíg i ≤ n^4 végezd el
      i ← 4 * i
    vége(amíg)
    j ← j DIV 2
  vége(amíg)
  visszatérít j
Vége(algoritmus)
```

A következő bonyolultsági osztályok közül melyikhez tartozik hozzá a fenti algoritmus időbonyolultsága?

a. $O(\log_2 n^2)$
b. $O(\log_2^2 n^2)$
c. $O(\log_4^2 n)$
d. $O(\log_2 \log_4 n)$

Helyes válasz:

Legyen a következő algoritmus, amelynek paramétere az n, nullától különböző természetes szám és amely egy természetes számot térít vissza.

```
Algoritmus f(n):
  j ← n
  Amíg j > 1 végezd el
    i ← 1
    Amíg i ≤ n^4 végezd el
      i ← 4 * i
    vége(amíg)
    j ← j DIV 2
  vége(amíg)
  visszatérít j
Vége(algoritmus)
```

A következő bonyolultsági osztályok közül melyikhez tartozik hozzá a fenti algoritmus időbonyolultsága?

a. $O(\log_2 n^2)$
b. $O(\log_2^2 n^2)$
c. $O(\log_4^2 n)$
d. $O(\log_2 \log_4 n)$

Helyes válasz: b, c

BÓNUSZ

Állapítsd meg az alábbi alprogram bonyolultságát n függvényében a Θ jelölést használva, ha a kezdeti hívás `B(1, n)`!

```
ALGORITMUS B(bal, jobb)
  HA (bal < jobb) akkor
    hossz = jobb - bal
    B(bal + hossz / 4, (bal + jobb) / 2)
    B(jobb - hossz / 4 + 1, jobb)
    i = 0
    AMÍG (hossz > i) végezd el:
      hossz = hossz / 3
      i = i / 2
    VÉGE(Amíg)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Helyes válasz:

BÓNUSZ

Állapítsd meg az alábbi alprogram bonyolultságát n függvényében a Θ jelölést használva, ha a kezdeti hívás `B(1, n)`!

```
ALGORITMUS B(bal, jobb)
  HA (bal < jobb) akkor
    hossz = jobb - bal
    B(bal + hossz / 4, (bal + jobb) / 2)
    B(jobb - hossz / 4 + 1, jobb)
    i = 0
    AMÍG (hossz > i) végezd el:
      hossz = hossz / 3
      i = i / 2
    VÉGE(Amíg)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Helyes válasz: $\Theta(\sqrt{n})$ (a Mester tétel alapján)

Bináris keresés

Feladat

Adott egy n egész számból álló szigorúan növekvő sorozat. Állapítsuk meg egy adott szám helyét a sorozatban! Ha az illető szám nem található meg a sorozatban, írjunk ki megfelelő üzenetet!

Példa: [1 4 5 11 12 13 25]
Ha a keresett szám a 13-as, visszatérítjük, hogy ez a 6. pozíción található.
Ha a keresett szám a 14-es, visszatérítjük, hogy ez nem található meg a sorozatban.
Megjegyzés: A bináris keresés módszere ennél általánosabb, használható a legkisebb elem helyének meghatározására, amely nagyobb mint a keresett elem, vagy a legnagyobb elem helyének a meghatározására, amely kisebb, mint a keresett elem. A sorozat lehet implicit is, például egy folytonos valós függvény.

Bináris keresés

Elemzés

Az elemet a sorozat közepén fogjuk először keresni, egyre csökkentjük az aktuális sorozatot, amelyben a keresést végezzük ennek a végeit a `bal` és `jobb` változók jelzik, kezdetben `bal = 1, jobb = n`. Három eset lehetséges:

1. Ha `keresett = a[közép]`, megtaláltuk az elemet a `közép` indexen.
2. Ha `keresett < a[közép]`, mivel a sorozat rendezett, az elemet az aktuális sorozat első felében keressük tovább a `bal..közép - 1` intervallumban.
3. Ha `keresett > a[közép]`, a keresett számot az aktuális sorozat második felében keressük tovább a `közép + 1..jobb` intervallumban.

Bináris keresés

Elemzés

- A feladat átalakul ugyan két részfeladattá, de ezek közül csak az egyiket kell megoldani.
- Így nem lesz szükség a *Divide et impera* utolsó lépésére, az eredmények összerakására.
- Ennek köszönhetően a bináris keresés iteratívan is könnyedén implementálható.
- Ha az aktuális sorozat üressé vált, azt jelenti, hogy a keresett elem nem található meg a bemeneti sorozatban.

Bináris keresés (rekurzívan)

Pszeudokód

```
ALGORITMUS BinKeresRek(bal, jobb, keresett, a)
  HA (bal > jobb) akkor
    VISSZATÉRÍT: -1 //a keresett elem nincs a sorozatban
  KÜLÖNBEN
```

Bináris keresés (rekurzívan)

Pszeudokód

```
    közép = (bal + jobb) / 2
    HA (keresett > a[közép]) akkor
      VISSZATÉRÍT: BinKeresRek(közép + 1, jobb, keresett, a)
    KÜLÖNBEN
      HA (keresett < a[közép]) akkor
        VISSZATÉRÍT: BinKeresRek(bal, közép - 1, keresett, a)
      KÜLÖNBEN
        VISSZATÉRÍT: közép
      VÉGE(Ha)
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Bináris keresés (iteratívan)

Pszeudokód

```
ALGORITMUS BinKeresIt(bal, jobb, keresett, a)
  bal = 1
  jobb = n
  megvan = HAMIS
```

Bináris keresés (iteratívan)

Pszeudokód

```
AMÍG ((NEM megvan) ÉS (bal <= jobb)) végezd el:
  közép = (bal + jobb) / 2
  HA (keresett > a[közép])
    bal = közép + 1
  KÜLÖNBEN
    HA (keresett < a[közép])
      jobb = közép - 1
    KÜLÖNBEN
      megvan = IGAZ
    VÉGE(Ha)
  VÉGE(Ha)
VÉGE(Amíg)
```

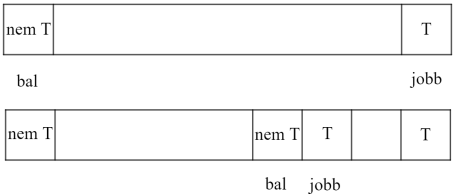
Bináris keresés (iteratívan)

Pszeudokód

```
  HA (NEM megvan) akkor
    VISSZATÉRÍT: -1
  KÜLÖNBEN
    VISSZATÉRÍT: közép
  VÉGE(Ha)
VÉGE(Algoritmus)
```

Általánosított (diszkrét) bináris keresés

- Legyen `T` egy tulajdonság melyet a tömb elemein értelmezünk.
- Feltételezzük, hogy kezdetben a `bal` indexen található elem nem rendelkezik a `T` tulajdonsággal, míg a `jobb` indexen található rendelkezik a `T` tulajdonsággal (`bal` $<$ `jobb`).
- Ekkor a bináris keresés ezen változata talál két szomszédos elemet (`jobb` $-$ `bal` $= 1$), úgy, hogy `T(bal) = HAMIS` és `T(jobb) = IGAZ`
- Vegyük észre, hogy az algoritmus akkor is helyesen működik, ha a sorozatban több mint egy „váltási pont” van.



Általánosított (diszkrét) bináris keresés

Pszeudokód

```
ALGORITMUS AltalanosBinKereses(bal, jobb, T)
  AMÍG (jobb - bal > 1)
    közép = (bal + jobb) / 2
    HA (T(közép))
      jobb = közép
    KÜLÖNBEN
      bal = közép
    VÉGE(Ha)
  VÉGE(Amíg)
VÉGE(Algoritmus)
```

Általánosított (diszkrét) bináris keresés

Példák

- A klasszikus bináris keresést úgy kapjuk az általánosból, hogy a tömb két végére strázsát állítunk (`a[0]` $= -\infty$, `a[n + 1]` $= \infty$), kezdetben `bal` $= 0$ és `jobb` $= n + 1$ és a tulajdonságot a következő módon vesszük fel: `T(i)` $=$ `a[i]` $\geq$ `keresett`
- Az algoritmus futása után a keresett index a `jobb` változóban lesz. Ha `a[jobb]` $\neq$ `keresett`, akkor a keresett elem nem szerepel a tömbben és visszatéríthetünk -1-et.

Bináris keresés C++-ban

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector <int> a = {1, 3, 3, 5};

    cout << lower_bound(a.begin(), a.end(), 3) - a.begin() << endl; //1
    cout << lower_bound(a.begin(), a.end(), 4) - a.begin() << endl; //3

    cout << upper_bound(a.begin(), a.end(), 3) - a.begin() << endl; //3
    cout << upper_bound(a.begin(), a.end(), 4) - a.begin() << endl; //3

    cout << equal_range(a.begin(), a.end(), 3).first - a.begin() << endl; //1
    cout << equal_range(a.begin(), a.end(), 3).second - a.begin() << endl; //3

    //C++20-ig
    cout << binary_search(a.begin(), a.end(), 3) << endl; //TRUE
    cout << binary_search(a.begin(), a.end(), 4) << endl; //FALSE
}
```

Ternáris keresés

- Egy olyan sorozat minimumát kereshetjük meg vele, amely előbb csökken, majd növekszik. Szimmetrikus módon egy olyan sorozat maximumát is, amely előbb növekszik, majd csökken.
- Három harmadra osztjuk az intervallumot és attól függően, hogy a két „harmadoló pontban" milyen értékek vannak, az egyik harmadot „eldobhatjuk".
- Akkor állunk meg, amikor az intervallum hossza elérte a három elemet, ekkor a megmaradt három elem között lesz a keresett érték.

Ternáris keresés

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector <int> a = {8, 1, 3, 4, 5};

    int bal = 0, jobb = a.size() - 1;
    while (jobb - bal > 2)
    {
        int k1 = bal + (jobb - bal) / 3;
        int k2 = bal + 2 * (jobb - bal) / 3;
        if (a[k1] < a[k2]) jobb = k2;
        else bal = k1;
    }
    cout << min({a[bal], a[bal + 1], a[jobb]}) << endl;
}
```

Legyen a `bűvös(x)` algoritmus, ahol x természetes szám ($1 \leq x \leq 32000$):

```
Algorithm bűvös(x):
    bal ← 1
    jobb ← x
    While bal ≤ jobb execute
        közép ← (bal + jobb) DIV 2
        If közép * közép = x then
            return True
        EndIf
        If közép * közép < x then
            bal ← közép + 1
        else
            jobb ← közép − 1
        EndIf
    EndWhile
    return False
EndAlgorithm
```

Állapítsátok meg, hogy az alábbi állítások közül melyek igazak!

a. Az x bármely 10-nél szigorúan kisebb értékére az algoritmus *False*-t térít vissza
b. Az algoritmus törzsténye zőkre bontja az x számot
c. Az algoritmus *True*-t térít vissza, ha az x szám négyzetszám.
d. Az algoritmus nem térít vissza *True*-t az x egyetlen megengedett értékére sem.

Helyes válasz:

Legyen a `bűvös(x)` algoritmus, ahol x természetes szám ($1 \leq x \leq 32000$):

```
Algorithm bűvös(x):
    bal ← 1
    jobb ← x
    While bal ≤ jobb execute
        közép ← (bal + jobb) DIV 2
        If közép * közép = x then
            return True
        EndIf
        If közép * közép < x then
            bal ← közép + 1
        else
            jobb ← közép − 1
        EndIf
    EndWhile
    return False
EndAlgorithm
```

Állapítsátok meg, hogy az alábbi állítások közül melyek igazak!

a. Az x bármely 10-nél szigorúan kisebb értékére az algoritmus *False*-t térít vissza
b. Az algoritmus törzstényezőkre bontja az x számot
c. Az algoritmus *True*-t térít vissza, ha az x szám négyzetszám.
d. Az algoritmus nem térít vissza *True*-t az x egyetlen megengedett értékére sem.

Helyes válasz: c

Jancsinak implementálnia kell a bináris keresés algoritmusát, amellyel meg kell keresnie egy a elemet az n elemű, egész számokat tároló, növekvően rendezett V sorozatban $(1 \leq n \leq 1000)(V[1], V[2], \ldots V[n])$. Jancsi a következő algoritmust írta:

```
Algoritmus binárisKeresés(a, n, V):
   bal ← 1
   jobb ← n
   Amíg jobb – bal > 1 végezd el
      közép ← (bal + jobb) DIV 2
      Ha a ≤ V[közép] akkor
         jobb ← közép
      különben
         bal ← közép
      vége(ha)
   vége(amíg)
   visszatérít jobb
Vége(algoritmus)
```

Döntsétek el, hogy a következő állítások közül melyek igazak:

- **a** Ha $n = 1$, akkor az algoritmus mindig 1-et térít vissza.
- **b** Bármely $n \geq 1$-re, ha a kisebb a sorozat minden eleménél, a Jancsi algoritmusa 1-et térít vissza.
- **c** Ha az a érték megtalálható a sorozatban, a Jancsi algoritmusa NEM mindig téríti vissza az a pozícióját (indexét a V vektorban).
- **d** Ha $n > 1$ és a nagyobb a sorozat minden eleménél, a Jancsi algoritmusa az n értékét téríti vissza.

Helyes válasz:

Jancsinak implementálnia kell a bináris keresés algoritmusát, amellyel meg kell keresnie egy a elemet az n elemű, egész számokat tároló, növekvően rendezett V sorozatban $(1 \leq n \leq 1000)(V[1], V[2], \ldots V[n])$. Jancsi a következő algoritmust írta:

```
Algoritmus binárisKeresés(a, n, V):
   bal ← 1
   jobb ← n
   Amíg jobb - bal > 1 végezd el
      közép ← (bal + jobb) DIV 2
      Ha a ≤ V[közép] akkor
         jobb ← közép
      különben
         bal ← közép
      vége(ha)
   vége(amíg)
   visszatérít jobb
Vége(algoritmus)
```

Döntsétek el, hogy a következő állítások közül melyek igazak:

- a) Ha $n = 1$, akkor az algoritmus mindig 1-et térít vissza.
- b) Bármely $n \geq 1$-re, ha a kisebb a sorozat minden eleménél, a Jancsi algoritmusa 1-et térít vissza.
- c) Ha az a érték megtalálható a sorozatban, a Jancsi algoritmusa NEM mindig téríti vissza az a pozícióját (indexét a V vektorban).
- d) Ha $n > 1$ és a nagyobb a sorozat minden eleménél, a Jancsi algoritmusa az n értékét téríti vissza.

Helyes válasz: a, c, d

Melyek lehetnek egy olyan vektor elemei, amelyre, ha alkalmazzuk a bináris keresés algoritmusát és a 36-os értéket keressük, az összehasonlítások egymás után rendre a 12, 24, 36 értékekkel történnek?

a) 2, 4, 7, 12, 24, 36, 50
b) 2, 4, 8, 9, 12, 16, 20, 24, 36, 67
c) 4, 8, 9, 12, 16, 24, 36
d) 12, 24, 36, 42, 54, 66

Helyes válasz:

Melyek lehetnek egy olyan vektor elemei, amelyre, ha alkalmazzuk a bináris keresés algoritmusát és a 36-os értéket keressük, az összehasonlítások egymás után rendre a 12, 24, 36 értékekkel történnek?

a) 2, 4, 7, 12, 24, 36, 50
b) 2, 4, 8, 9, 12, 16, 20, 24, 36, 67
c) 4, 8, 9, 12, 16, 24, 36
d) 12, 24, 36, 42, 54, 66

Helyes válasz: b, c

Legyen a `verifica(n, p1, p2)` algoritmus, ahol n, $p1$ és $p2$ természetes számok ($1 \leq n, p1, p2 \leq 10^6$).

```
Algorithm verifica(n, p1, p2):
    bt ← (p1 + p2) DIV 2
    If p1 > p2 then
        Return False
    EndIf
    If bt * bt = n then
        Return True
    EndIf
    If bt * bt > n then
        Return verifica(n, p1, bt - 1)
    EndIf
    Return verifica(n, bt + 1, p2)
EndAlgorithm
```

A következő állítások közül melyek igazak?

a. Ha $p1, p2$ és n relatív prímek, akkor a `verifica(n, p1, p2)` hívás *True*-t térít vissza.

b. Az algoritmus a bináris keresés módszerét alkalmazza és ha n prímszám, a `verifica(n, 1, n)` hívás True-t *térít* vissza.

c. A `verifica(n, 1, n)` hívás akkor és csakis akkor térít vissza *True*-t, ha az n négyzetszám.

d. Ha $p1 \leq n \leq p2$ és a $[p1, n]$ és $[n, p2]$ intervallumokban létezik legalább egy-egy négyzetszám, akkor a `verifica(n, p1, p2)` hívás *True*-t térít vissza.

Helyes válasz:

Legyen a `verifica(n, p1, p2)` algoritmus, ahol n, $p1$ és $p2$ természetes számok ($1 \leq n, p1, p2 \leq 10^6$).

```
Algorithm verifica(n, p1, p2):
    bt ← (p1 + p2) DIV 2
    If p1 > p2 then
        Return False
    EndIf
    If bt * bt = n then
        Return True
    EndIf
    If bt * bt > n then
        Return verifica(n, p1, bt - 1)
    EndIf
    Return verifica(n, bt + 1, p2)
EndAlgorithm
```

A következő állítások közül melyek igazak?

a. Ha $p1, p2$ és n relatív prímek, akkor a `verifica(n, p1, p2)` hívás *True*-t térít vissza.

b. Az algoritmus a bináris keresés módszerét alkalmazza és ha n prímszám, a `verifica(n, 1, n)` hívás True-t *térít* vissza.

c. A `verifica(n, 1, n)` hívás akkor és csakis akkor térít vissza *True*-t, ha az n négyzetszám.

d. Ha $p1 \leq n \leq p2$ és a $[p1, n]$ és $[n, p2]$ intervallumokban létezik legalább egy-egy négyzetszám, akkor a `verifica(n, p1, p2)` hívás *True*-t térít vissza.

Helyes válasz: c

Legyen a következő játék: az egyik játékos (Játékos1) gondol egy 1 és 1000 közötti természetes számra. A másik játékosnak (Játékos2) ki kell találnia ezt a számot minél kevesebb próbálgatással. A titkos szám „birtokosa" (Játékos1) egy-egy találgatásra csak annyit válaszol, hogy a titkos szám kisebb vagy nagyobb mint a másik játékos (Játékos2) által feltételezett szám.
Írjunk programot amely a fenti játékot szimulálja, úgy, hogy a programunk a Játékos2, azaz ő találja ki a felhasználó titkos számát. Nem tippelhetünk több mint 10-szer.

Adott az $a[1], a[2]....a[n] (n \geq 1)$ különböző természetes számokat tartalmazó csökkenő sorozat és az x természetes szám. Írjatok algoritmust pszeudokódban, melynek időbonyolultsága $O(log_2 n)$ és amely visszatéríti azt a *poz* pozíciót, amelyen az x elem megjelenik az a sorozatban (ha x megjelenik a sorozatban), vagy azt a pozíciót, amelyre x-et be kellene szúrni az a sorozatba úgy, hogy az elemek csökkenő sorrendje megmaradjon. Indokoljátok az algoritmus bonyolultságát. **Megjegyzés:** Egy elem beszúrása a k. pozicióra feltételezi a $k, k+1, ..., n$ pozíciókón lévő elemek egy pozícióval való jobbra tolását és a sorozat hosszának növelését.

Példa. Ha $n = 4$ és az a sorozat (14, 12, 9, 5) akkor:

- ha $x = 15$ a visszatérített érték *poz*=1, mivel a 15 beszúrása után a sorozat (15, 14, 12, 9, 5);
- ha $x = 11$ a visszatérített érték *poz*=3, mivel a 11 beszúrása után a sorozat (14, 12, 11, 9, 5);
- ha $x = 12$ a visszatérített érték *poz*=2, mivel 12 már megjelenik a 2. pozíción.

Legyen két sorozat, amelyeknek elemei különböző természetes számok: az a sorozat elemeinek száma n ($0 < n \leq 100\ 000$), a b sorozat elemeinek száma m ($0 < m \leq 100\ 000$) és növekvően rendezett. Határozzuk meg azt a c sorozatot amely a két sorozat minden közös elemét egyszer tartalmazza.

Példa

Bemenet	Kimenet
4 5 -7 -2 3 5 -2 3 5 7 8	3 5 -2 3

Adott egy n elemű tömb, mely 32 bites előjeles egész számokat tartalmaz és egy x 32 bites előjeles egész szám. Határozzuk meg, hogy létezik-e két olyan eleme a tömbnek, melyek összege pontosan x. Alkalmazzunk bináris keresést! Ha több megoldás létezik, írjuk ki a lexikografikusan elsőt

Példa

Bemenet	Kimenet
5 12 1 2 16 4 8	1 4 5

Adott n elemű a sorozat, melyre igaz, hogy $a_1 > a_2$ és $a_{n-1} < a_n$ ($3 \leq n \leq 1000$, $a_i \neq a_{i+1}, \forall i = \overline{1, n-1}$). *Lokális minimum*nak nevezünk minden olyan a_i elemet, amelyre $a_{i-1} > a_i < a_{i+1}$ és $i = \overline{2, n-1}$. Határozzunk meg egy tetszőleges lokális minimumot egy tömbben, melynek kezdetben ismeretlenek az elemei! Nem kérhetjük le több mint 25 elem értékét.

Egy f függvény gyökének azt az x pontot nevezzük, amelyre $f(x) = 0$. Egy f függvény lineáris, ha $f(x) = ax + b$ formában írható, ahol $a, b \in \mathbb{R}, a \neq 0$. Határozzuk meg *oszd meg és uralkodj* módszerrel egy lineáris függvény x gyökét 10^{-6} pontossággal, tudván, hogy $x \geq 0$. Nem kérhetjük le a függvény értékét több, mint 100-szor.