Прва београдска гимназија
Београд, Цара Душана 61

Матурски рад из програмирања

АЛГОРИТАМСКА ТЕХНИКА: ПОДЕЛИ ПА ВЛАДАЈ

Ментор:
Проф. Милош Пушић

Ученик:
Милош Ракић IV_{10}

Београд, јун 2022

ПРВА БЕОГРАДСКА ГИМНАЗИЈА

Садржај

1. Увод

Стратегија подели па владај, кроз историју познатија и као „завади па владај“ (Лат: *Divide et impera*), најчешће је примењивана у областима политике и социологије. Сама изрека се приписује Јулију Цезару који је овом стратегијом разједињавао своје противнике и носио се са знатно мањим проблемима који му се, тако слаби, нису могли супроставити. У прошлости ова тактика је примењивана на разне начине. Велики број европских владара, од француског Луја XI па све до каснијих Хабзбурга, подстицали су или неговали свађе између мањих група својих непријатеља, не би ли их полако поражавали, једног по једног. Овај тип политичке манипулације захтева изузетно добро познавање људи којима се манипулише. Касније је се метода *подели па владај* проширила на економију па и саму политику. Истискање конкуренције на тржишту као и слабљење политичке опозиције само су једни од мноштва примера примене чувене стратегије.

Како се у програмирању уместо људима манипулише подацима, метода *подели па владај* се, сходно сврси, појављује у виду алгоритма, односно, алгоритамске технике. Алгоритам осмишљен тако да задати проблем рекурзивно разбија на мање подпроблеме који се на исти начин поједностављују, а затим тако прости решавају, представља саму суштину Цезарове мисли.

Решавање задатака овом методом даје прилично ефикасне резултате. Моћ уочавања мањих проблема унутар једног великог стиче се вежбом и искуством. Наравно, сваки се проблем може посматрати на наиван и површан начин, зато се на разним интервјуима најчешће појављују наизглед једноставни задаци где ће кандидат на основу начина приступа проблему показати колико је вешт. „Алгоритам *подели па владај* је један од оних алгоритама код којих се вештост кандидата који га користе уопште не доводи у питање.“[1]

[1] Проф. Ерик Гримсон, лекција 10: Divide and Conquer methods

2. Рад алгоритма

2.1. Подели

„Подели“ је први корак у имплементацији овог алгоритма. На рекурзиван начин, велики проблем се дели на мање и једноставније проблеме. Битно је напоменути да новонастали подпроблеми, иако знатно мањи, и даље на неки начин представљају оно од чега смо кренули. Рекурзивна функција позива саму себе са све мањим аргументима, односно аргументима који све више теже некој граничној вредности односно, општем случају. Тај процес назива се *рекурзивни случај*. Што значи, када се *подели* алгоритам имплементира, одређује се *рекурзивни случај* који ће упрошћавати тренутни проблем.

Овај корак је могуће осмислити и на итеративан начин али се он ретко када користи. Проблем настаје у каснијим корацима када треба искористити добијене резултате. Брзина извршења програма би била непромењена, али количина меморије коју итеративан начин захтева је знатно већа. Стога, *подели* корак готово увек позива рекурзивна функција.

2.2. Владај

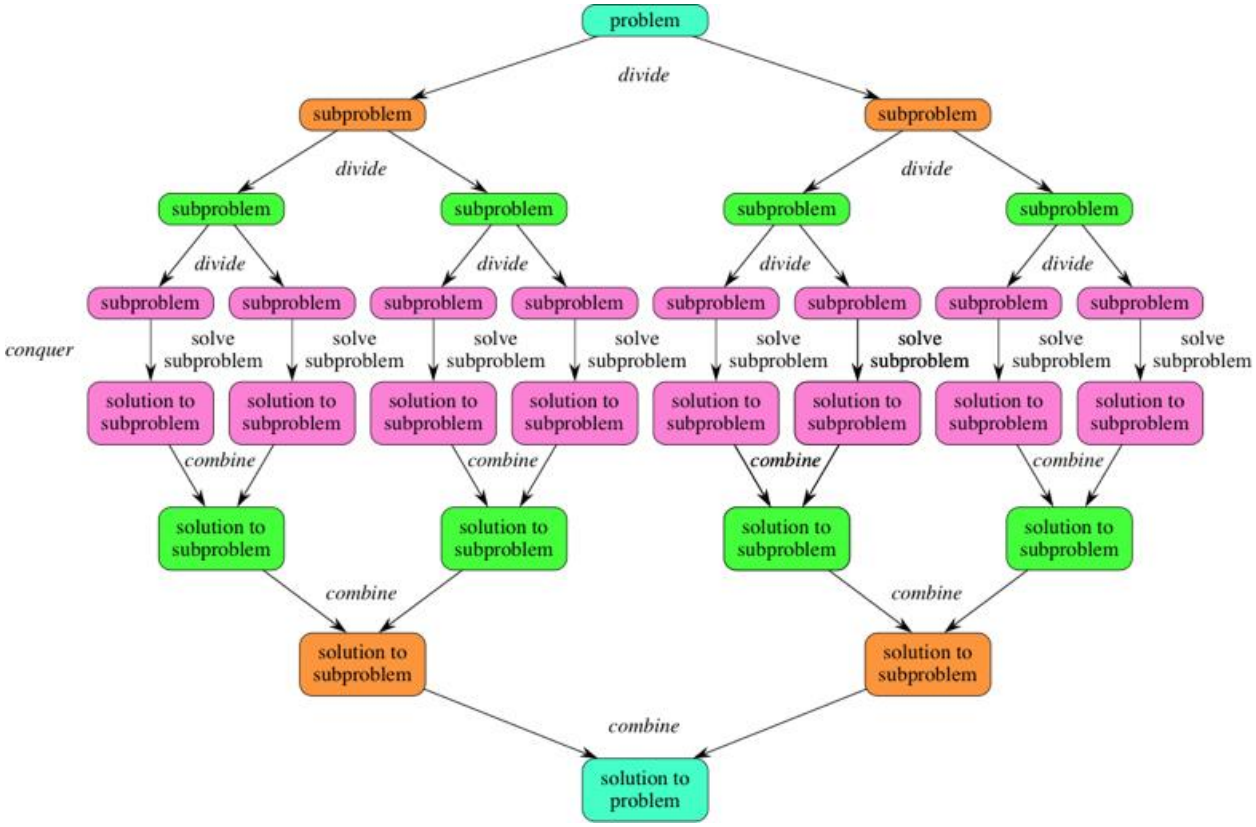
Затим имамо корак под називом „владај“. Овде се сви настали подпроблеми решавају. Како смо до сада почетни задатак *рекурзивним случајевима* поделили на најмање могуће елементе, решавамо их користећи основне логичке и рачунске операције.

Имплементација овог корака такође се налази у рекурзивној функцији, а алгоритам *владај* уствари представља општи случај рекурзивне фукнције. Односно, када проблеми постану довољно мали да је од њих немогуће направити следећи *рекурзивни случај,* кажемо да је рекурзија „дошла до дна“. Тада се решава њен општи случај и свака малопређашње позивана функција добија своју вредност. Другим речима, сваки настали подпроблем који представља општи случај рекурзивне функције, у овом кораку, имаће своје решење.

2.3. Споји

Код „споји“ корака, сва решења подпроблема се комбинују не би ли решила цео почетни проблем. Излаз функција које су решиле свој општи случај, постаје улаз функција са већим подпроблемом. Како нам се рекурзија поново „пуни“, комбинујемо решења подпроблема истог нивоа, да би на излазу

проследили одговарајућ параметар функцији на вишем нивоу. Последњи излазни податак представља решење задатог проблема.



Слика 1: Елементарни приказ принципа рада алгоритамске технике подели па владај

2.4. Временска сложеност

Са слике 1 јасно видмо да број подпроблема чини једну геометријску прогресију а њихова решења геометријску регресију. Временска сложеност проблема који се решавају овом техником не може да се генерализује али у даљим примерима ћемо увидети да је скоро увек у питању логаритамска сложеност.

Рецимо да је T(n) време решавања датог проблема величине n. Претпоставимо да се проблем дели на k општих подпроблема и да имамо укупно a подела. Сви подпроблеми решили би се за време aT(n/k). Нека је D(n) време дељења почетног проблема на подпроблеме, а C(n) време спајања свих решења. Добијамо елементарну једначину:

$$T(n) = aT\left(\frac{n}{k}\right) + D(n) + C(n)$$

3. Примери коришћења технике

У следећих 6 примера рад програма који користе алгоритамску технику *подели па владај* биће јасно илустрован. Кроз дате примере уочаваћемо ефикасност методе, предности и мане коришћене технике и поредићемо је са наивним приступом свим примерима.

3.1. Алгоритам бинарне претраге

Текст задатка: Дат је *сортиран* низ од *n* елемената. Одредити да ли у датом низу постоји елемент вредности *x*.

Улаз1: niz[] = [1,5,6,7,13,15,19] **Улаз2: niz[] = [1,5,6,7,13,15,19]**

***x* = 7** ***x* = 10**

Излаз1: Елемент пронађен на индексу 3.
Излаз2: Елемент није пронађен

Наивни приступ овом проблему подразумева линеарну претрагу датог низа. Редом проверавање сваког елемента све док елемент *x* није пронађен. Временска сложеност ове претраге је O(n) где је n величина датог низа. Овај приступ не узима у обзир то што је низ *сортиран*, самим тим знатно се губи на времену.

Идеја алгоритма бинарне претраге долази од технике *подели па владај*. Како је описано у *раду алгоритма (2),* дати низ поделићемо на два мања подниза и њима ћемо рекурзивно управљати. Али, уместо да радимо са оба низа, отарасићемо се једног и користити само други. Таква одлука доноси се у само једном поређењу. Дакле, бинарана претрага смањује свој простор за претрагу за пола након сваког корака. На почетку, простор за претрагу представља цео низ, а бинарна претрага одлучује који од два новодобијена подниза узима, корситећи својство да је унети низ *сортиран*. То ради тако што пореди средњи елемент са траженим елементом *x*. Ту долазимо до 3 могућа слулаја:

1. `x = niz[sredina]` -> Штампамо тражени индекс, ондосно вредност `sredina`.
2. `x > niz[sredina]` -> Одбацујемо леви подниз, тражимо средину десног, па га опет делимо на два дела.
3. `x < niz[sredina]` -> Одбацујемо десни подниз, тражимо средину левог, па га опет делимо на два дела.

Процес се понавља све док елемент *x* није пронађен.

Временска сложеност овог алгоритма далеко је мања од линеарне. Пошто је сваки следећи подниз дупло мањи од прошлог, све док у најгорем случају није величине 1, добијамо логаритамску сложеност O($log_2 n$).

Имплементација алгоритма бинарне претраге:

```
int binarnaPretraga(int niz[], int manji, int veci, int x)
  {
      //Opsti slucaj (Prostor za pretragu ne postoji)
      if (manji > veci)
      {
          return -1;
      }

      //Pronalazenje srednjeg elementa niza i
      //poredjenje sa elementom x
      int sredina = (manji + veci) / 2;

      //Opsti slucaj (Trazena vrednost je pronadjena)
      if (x == niz[sredina])
      {
          return sredina;
      }

      //Uklanjanje svih elemenata desno
      //ukljucujuci i srednji element
      else if (x < niz[sredina])
      {
          return binarnaPretraga(niz, manji, sredina - 1, x);
      }

      //Uklanjenje svih elemenata levo
      //ukljucujuci i srednji element
      else
      {
          return binarnaPretraga(niz, sredina + 1, veci, x);
      }
  }
```

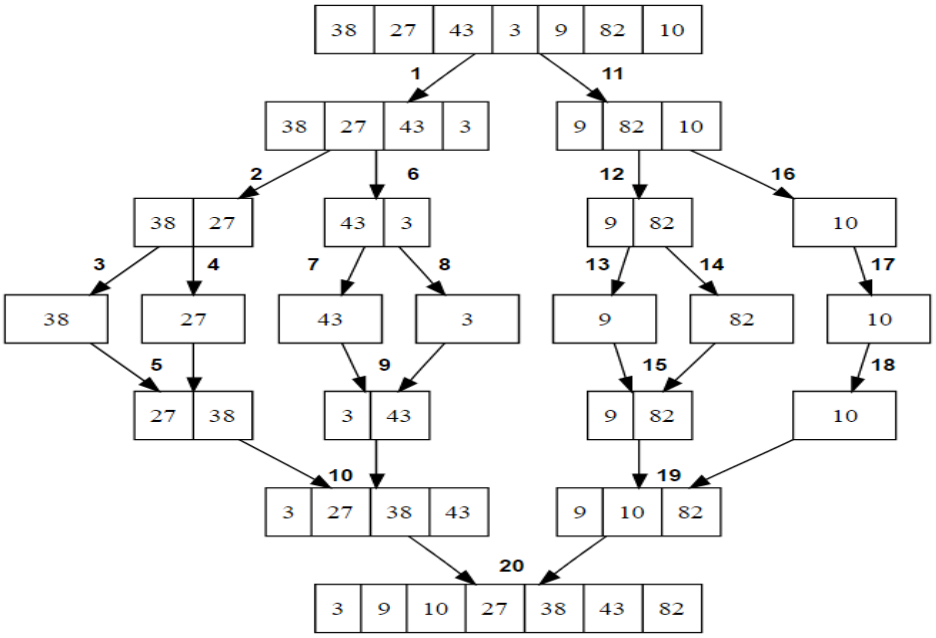
Аргументи рекурзивне функције су редом: Дати *сортирани* низ, односно касније подниз, у којем се претражује задати елемент, индекс најмањег елемента подниза, индекс највећег елемента подниза и на самом крају, тражени елемент.

Како је бинарна претрага делеко ефикаснија него линеарна, она представља типичан пример коришћења алгоритамсе технике *подели па владај*. Јасно се види њена предност (ефикасност), али можемо приметити да је њено коришћење ограничено (захтева *сортиран* низ).

3.2 Merge sort алгоритам

Merge sort је један од ефикаснијих алгоритама за сортирање низова било којих елемената који могу да се пореде логичким операцијама. Цео алгоритам заснован је на техници *подели па владај.* Дати несортирани низ дели се на поднизове (што приближније) једнаке дужине који се рекурзивно сортирају.

Алгоритам подразумева дељење датог низа дужине n на n поднизова дужине 1 (низ дужине један се сматра сортираним), и непрекидно спајање сортираних поднизова не би ли се добили већи сортирани поднизови. Наравно, новодобијени низ дужине n ће представљати сотирани почетни низ, или другим речима, решење проблема.



Слика 2: Merge sort алгоритам низа дужине 7

Код који буде представљао овај алгоритам управо ће се и састојати из два дела. Фунцкија која спаја 2 сортирана подниза (сл. 4), и функција која дели прослеђени (под)низ на два дела све док не дође до сортираног низа (сл. 3). Управо у друго поменутој функцији налазиће се општи случај.

Претпоставимо да морамо да сортирамо низ niz[], рецимо да први елемент датог низа има индекс *p*, а последњи елемент се налази на месту *r*. То ћемо представити као niz[*p*..*r*]. У „подели“ кораку морамо преполовити дати низ. Рецимо да средишњи елемент има индекс *q*, што значи да сада наш niz[*p*..*r*] делимо на низове niz[*p*..*q*] и niz[*q*+ 1..*r*]. Код „владај“ корака, покушаћемо да сортирамо два новодобијена подниза. Ако нисмо дошли до општег случаја (подниз дужине 1), делимо оба подниза на исти начин и поново покушавамо да их соритамо. Када рекурзивна функција стигне до свог општег случаја, на ред долази „споји“ корак. Врши се спајање два сортирана подниза и добијање већег сортираног (под)низа.

```
mergeSort(niz[], p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(niz[], p, q)
    mergeSort(niz[], q+1, r)
    merge(niz[], p, q, r)
```

Слика 3: Приказ псеудо кода merge sort алгоритма

Сваки рекурзивни алгоритам зависи од свог општег случаја и могућности да користи информације извучене из њих. Најбитни део merge sort алгоритма је управо та функција merge(…). Као што је већ напоменуто, функција merge(…) ће спојити два сортирана подниза. Њој се прослеђују подаци о индкесима почетка оба подниза, а функција је задужена да памти тренутни индекс коначно сортираног низа, као и тренутне индексе поднизова.

Поднизови се спајају на следећи начин. Како су оба сортирана, поредићемо први елемент једног и другог подниза. Онај који је мањи штампамо у коначан, спојени, низ и прелазимо на следеће поређење. Пореде се следећи члан низа са претходно мањим елементом и претходно већи елемент. Опет, мањи се штампа и поређење се наставља. Битно је напоменути да пре сваког поређења треба проверити да ли се неки од поднизова испразнио. Ако јесте, остатак елемената другог низа је потребно

само преписати у коначни низ и позвати return функције merge(…).

Можемо видети (сл. 4) да се само спајање поднизова након константног дељења почетног низа своди на пар основних рачунских операција. Ово објашњава тврђење са почетка поглавља да се на овај начин могу сортирати сви подаци који могу логички да се пореде. Такође је речено да је merge sort један од ефикаснијих алгоритама, што значи да некада и није прави избор за соритрање ваших података. Временска сложеност овог алгоритма је у сваком случају (worst case и best case) O(n*log n), што говори да ће се дељење као и спајање обављати у целости чак и ако је низ сортиран. Са друге стране, овај приступ је најефикаснији када баратамо са већом количином података, што нам и логаритамска сложеност говори. Још један мањак merge sort-а је тај што користи већу количину меморије, па може успорити рад машине на којој се извршава у каснијим процесима спајања. Та додатна количина произилази из прављења другог подниза, што на пример, код heap sort-а, није случај. Све у свему, глава предност овог алгоритма је та што је он *стабилан.*

```
void merge(int niz[], int p, int q, int r)
{
    //Velicine podnizova i njihova deklaracija
    int n1 = q - p + 1;
    int n2 = r - q;
    int niz1[n1], niz2[n2];
    for (int i = 0; i < n1; i++)
        L[i] = niz[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = niz[q + 1 + j];

    //Pamcenje trenutnih indeksa podnizova kao i konacnog niza
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Sve dok ne stignemo do kraja jednog od podnizova, birati manji element
    // pri poredjenju i staviti na tacnu poziciju u niz[p..r]
    while (i < n1 && j < n2)
    {
        if (niz1[i] <= niz2[j]){
            niz[k] = niz1[i];
            i++;
        }
        else{
            niz[k] = niz2[j];
            j++;
        }
        k++;
    }

    // Kada nam nestane elementa iz jednog od podnizova
    // uzeti ostale elemente i staviti ih na kraj niza niz[]
    while (i < n1){
        niz[k] = niz1[i];
        i++;
        k++;
    }

    while (j < n2){
        niz[k] = niz2[j];
        j++;
        k++;
    }
}
```

Слика 4: Приказ кода merge(...) функције

3.3. Највећи збир елемената подниза датог низа

Текст задатка: Дат је низ дужине n. Нека је променљива *x* збир свих елемената било ког подниза датог низа. Одредити највећу вредност променљиве *x*.

Улаз: niz[] = [2,-4,1,9,-6,7,-3] **Улаз: niz[] = [5,6,8,-9,-7,4,2]**

Излаз: *x* = 11 **Излаз: *x* = 19**

Наравно, наивно решење овог проблема подразумева проверавање сваког могућег подниза, рачунање збирова њихових елемената и памћење максимума. Овакав приступ даје нам временску сложеност од $O(n^2)$, због постојања две угњеждене петље, што је за веће вредности броја n превелика количина времена. Поставља се питање како написати ефикаснији програм?

Идеја је користити технику *подели па владај*. Наиме, као и до сада, дати низ поделићемо на два подниза једнаке дужине. „Подели“ корак затим обухвата даље рекурзивно дељење поднизова, док у „владај“ кораку израчунавамо највећи збир елемената подниза добијених поднизова. Другим речима, решавамо исти проблем користећи прослеђени подниз. Битно је напоменути да један од елемената збира левог подниза буде средишњи елемент прослеђеног (под)низа, док један од елемената збира десног подниза мора бити елемент који иде након средишњег елемента прослеђеног (под)низа. Овај детаљ је јако битан јер се на крају рекурзивне функције пореде три вредности: Највећи збир левог подниза, највећи збир десног подниза и сам збир 2 поменуте вредности. Последња вредност се мора укључити јер у супротном није укључен највећи збир који прелази преко средишњег елемента.

Значи, прослеђени низ у рекурзивној функцији се дели на два подниза једнаке дужине. Пре тога се проверава да ли прослеђени низ садржи један или ни један елемент (*општи случај*). У том случају функција враћа један елемент (јер је једини подниз прослеђеног низа баш тај елемент). Са друге стране, ако дати низ има 2 или више елемента, израчунавају се највећи збирови елемената новодобијених поднизова.

Приметимо (сл. 5) да су аргументи функције поред прослеђеног (под)низа и индекси првог и последњег члана тог (под)низа. Такође се може јасно видети да се код рачунања највећег збира елемената левог и десног подниза, почиње од средишњег елемента.

Након што смо пронашли највеће збирове са леве и десне стране, следи поновно дељење, али сада прво левог подниза а затим десног. Функција која тражи максималне збирове њихових поднизова биће позивана све док прослеђени подниз не буде имао један или ни један елемент. Пошто су максимуми леве и десне стране пронађени, на реду је тражење највеће вредности која прелази преко средишњег елемента. Она се једноставно добија када се две добијене вредности саберу. Највећа од три представља решење проблема.

Временска сложеност овог алгоритма је знатно боља од квадратне сложености наивног приступа (O(n*log n)),али се може још више побољшати. Линеарна сложеност је достижна путем динамичког програмирања.

```
int najveciZbir(int niz[], int manji, int veci)
{
    // Ako (pod)niz sadrzi 0 ili 1 element
    if (veci <= manji)
    {
        return niz[manji];
    }

    // Pronalazenje srednjeg elementa
    int srednji = (veci + manji) / 2;

    // Pronalazenje najveceg zbira u levom podnizu,
    // ukljucujuci srednji element
    int leviZbir = INT_MIN;
    int zbir = 0;
    for (int i = srednji; i >= manji; i--)
    {
        zbir += niz[i];
        if (zbir > leviZbir)
        {
            leviZbir = zbir;
        }
    }

    // Pronalazenje najveceg zbira u desnom podnizu,
    // ne ukljucujuci srednji element
    int desniZbir = INT_MIN;
    zbir = 0;    // reset sum to 0
    for (int i = srednji + 1; i <= veci; i++)
    {
        zbir += niz[i];
        if (zbir > desniZbir)
        {
            desniZbir = zbir;
        }
    }

    //Rekurzivno pronalazenje maksimuma levog
    //i desnog podniza trenutno prosledjenog (pod)niza
    int max_leviPodniz = najveciZbir(niz, manji, srednji);
    int max_desniPodniz = najveciZbir(niz, srednji + 1, veci);

    // Vracanje najveceg od 3
    return max(max(max_leviPodniz, max_desniPodniz), leviZbir + desniZbir);
    }
```

Слика 5: Рекурзивна функција за проналажење највећег збира елемената датог низа

3.4 Медијана 2 сортирана низа

Текст задатка: Дата су два сортирана низа, A[n] и B[m]. Пронаћи медијану сортитаног низа насталог спајањем A[] и B[].

Наиван и најлогичнији приступ проблему био би спојити два унета низа. Решење након тога не представља никакав напор. Израчунати индекс средишњег елемента (или два средишња елемента ако је n mod 2 = 0) и одштампати медијану. Процес спајања два сортирана низа приказан је у поглављу 3.2 (сл. 4) где видимо да би временска сложеност таквог поступка била O(m + n). Овај проблем могуће је решити ефикасније, уз помоћ технике *подели па владај* и алгоритма *бинарне претраге* (3.1), „велико O“ износило би O(log(m+n)). При размишљању о ефикаснијем решењу задатка, друго питање се намеће. На који начин програму дати до знања како да посматра два дата низа која треба спојити?

Практично гледано, нас занима само лева половина „спојеног“ низа, зато што се баш на последњем месту тог левог подниза налази тражена медијана. Наиме, низови A[] и B[] могу „допринети стварању“ спојеног низа на разне начине. Допринос у овом случају значи да поменути низови могу разним редоследом дати своје елементе. Ни један случај није искључен. На пример, прва половина спојеног низа може садржати само елементе низа A[], или можда само два од n елемената. Познавање које цифре учествују у стварању прве половине низа може нам помоћи да решимо проблем. Једноставно упоређивање последњег броја узетог из низа A[] и последњег броја узетог из низа B[] доводи нас до медијане, односно решења. Проблем такође неће представљати ни случај ако је m+n паран број. Само ће нам бити потребна следећа вредност у спојеном низу након посматраног подниза не би ли израчунали медијану. Коришћење бинарне претраге на први поглед није толико очигледно, али баш ће нам бинарана претрага помоћи при решавању овог проблема.

Јасно је да нам је треунти циљ да пронађемо вредности елемената оба низа A[] и B[] који учествују у првој половини спојеног низа A∪B. Ово нас води на даље упрошћавање проблема, како и сама техника *подели па владај* налаже. Пошто знамо број елемената који се појављује у првој половини спојеног низа ((m + n) / 2), можемо рећи да нас само занима број елемената из низа A[] који се појављује у тој половини. Рецимо

да је то број k, онда нам је и познат број елемената из низа B[] (((m + n) / 2) - k).

Узмимо један прост пример за илустрацију досадашње приче. Рецимо да је A[]={4,20,32,50,55,61} и B[]={1,15,22,30,70}. Спојени низ износио би A∪B={**1**,**4**,**15**,**20**,**22**,**30**,32,50,55,61,70}. Приметимо да је A допринео левој половини спојеног низа са 2 елемента, док је B дао своја 4. Такође се јасно види да је медијана спојеног низа баш последњи елемент прве половине тог низа. Пре самог спајања и посматрања A∪B, могли смо претпоставити да ће, на пример, оба низа допринети са 3 елемента (јер већ знамо да нам је потребно њих 6). Упоредићемо два последња елемента којима се доприноси поднизу,32 и 22, установити да је 32 већи и извести закључак да је управо 32 медијана спојеног низа. Ово наравно није случај. Значи 32 није било потребно поредити са 22 зато што је тај елемент низа A такође већи од следећег елемента низа B, 30, што значи да ће баш тај број „погурати“ 32 за једну позицију десно у сортираном спојеном низу, избацивајући га из прве половине. Сада ћемо „завладати“ проблемом и коснтатовати да је немогуће да допринос спојеном низу буде 3-3, пошто је четврти елемент низа B мањи од трећег елемента низа A, што нас води до претпоставке да допринос мора бити 2-4. Порцес је исти, поредимо други елемент првог и четврти елемент другог низа. Примећујемо да је 30 веће од 22 и закључујемо да је 30 медијана спојеног низа што и јесте тачно.

Сада нам се намеће ново питање, а то је како да утврдимо који је прави однос елемената који ће учествовати у првој половини спојеног низа? Одговор на ово питање је управо алгоритам бинарне претраге. Значи, рекли смо да низ A[] има n, а низ B[] има m елемената. То нам даље говори да они могу да се поделе на n и m делова. Претпоставимо да први низ има мању дужину, стога он најмање може допринети стварању спојеног низа са 0, а највише са n елемената (Зато што је n<(m + n)/2, јер је n<m). Те променљиве ћемо означити са aMin = 0 и aMax = n. У сваком кораку ћемо претпоставити да низ A[] учествује са **brojA= (aMin + aMax) / 2** елемента у стварању спојеног низа. Јасно је да ће низ B[] онда дати своја **brojB = ((m + n) / 2) – brojA** елемента. Како смо рекли у горњем примеру, исипутјемо већи од два елемента датих низова који се налазе на добијеним позицијама. Претпостављамо да је већи од њих медијана. Рецимо да је *x* елемент низа A[] на месту **brojA,** а *y* елемент низа B[] на месту **brojB**. Такође, узмимо да су *x'* и *y'* елементи датих низова који

долазе након елемената *x* и *y* респективно. Издвајамо 4 могућа случаја:

1. `y < x < y' ->` Пронашли смо решење задатка, *x* је медијана спојеног низа.
2. `x < y < x' ->` Пронашли смо решење задатка, *y* је медијана спојеног низа.
3. `x > y' ->` Значи да низ A[], доприноси са мање елемената него **brojA**, па је максимална вредност којом он доприноси спојеном низу мања. Другим речима aMax = **brojA** – 1.
4. `y > x' ->` Значи да низ B[], доприноси са мање елемената него **brojB**, па је максимална вредност којом он доприноси спојеном низу мања. Другим речима aMin = **brojA** + 1.

Следи поновно израчунавање променљивих **brojA** и **brojB,** све док први или други случај не буду испуњени. Када нађемо праве елементе *x* и *y,* остаје проверити да ли спојени низ има паран или непаран број елемената. Као што је већ речено, ако је тај број непаран, враћа се већи од *x* и *y*. У супротном потребно је пронаћи мањи од *x'* и *y'*, односно пронаћи први елемент у десној половини спојеног низа. Средња вредност мањег од *x'* и *y'* и већег од *x* и *y* представља решење задатка.

При кодирању решења датог проблема (сл. 6), битно је пазити на индексирање низова. Лако се може десити да изађемо ван граница датих низова па и то треба проверавати уз проверу који случај је заступљен.

```
public double pronadjiMedijanu(int[] A, int[] B)
        {
            int aDuzina = A.Length;
            int bDuzina = B.Length;
            // Budimo sigurni da uvek koristimo niz manje duzine
            if (aDuzina > bDuzina)
            {
                Swap(ref A, ref B);
                Swap(ref aDuzina, ref bDuzina);
            }
            int duzinaPolovineSpojenogNiza = (aDuzina + bDuzina + 1) / 2;

            // Posto znamo da je A kraci niz,
            // on moze doprineti spojenom nizu sa 0 ili aDuzina elemenata.
            int aMin = 0;
            int aMax = aDuzina;

            while (aMin <= aMax)
            {
                int brojA = (aMin + aMax) / 2;
                int brojB = duzinaPolovineSpojenogNiza - brojA;

                if (brojA > 0 && A[brojA - 1] > B[brojB]) //slucaj 3.
                {
                    aMax = brojA - 1;
                }
                else if (brojA < aDuzina && B[brojB - 1] > A[brojA]) //slucaj 4.
                {
                    aMin = brojA + 1;
                }
                else
                {
                    // Pronadjeni su tacni brojevi x i y.
                    // I dalje ne znamo kako se porede dva dobijena elementa.

                    int krajLevePolovine =
                        // Za slucaj da A ne ucestvuje
                        (brojA == 0)
                            // Poslednji element je y
                            ? B[brojB - 1]
                            // Za slucaj da B ne ucestvuje (jedino moguce ako je m == n)
                            : (brojB == 0)
                                // Poslednji element je x
                                ? A[brojA - 1]
                                //u suprotnom vracanje veceg elementa
                                : Math.Max(A[brojA - 1], B[brojB - 1]);

                    //ako je m+n neparno, medijana je veci element.
                    if (IsOdd(aDuzina + bDuzina))
                    {
                        return krajLevePolovine;
                    }

                    // Ako je m+n parno, da bismo izracunali medijanu potrebno je naci
                    // prvi element desne polovine, koji ce biti manji od elemenata A[x] i B[y].
                    int pocetakDesnePolovine =
                        //Za slucaj da su svi iz A[] iskorisceni
                        (brojA == aDuzina)
                            // Prvi desni je prvi iz B[], jer je y=0
                            ? B[brojB]
                            // Za slucaj da su svi iz B[] iskorisceni
                            : (brojB == bDuzina)
                                // Prvi desni je prvi iz A[], jer je x=0
                                ? A[brojA]
                                // u suprotnom trazenje minimuma
                                : Math.Min(A[brojA], B[brojB]);
                    return (krajLevePolovine + pocetakDesnePolovine) / 2.0;
                }
            }
            return -1; //Nepravilno unet niz
        }
```

Слика 5: Код решења проблема који тражи медијану два сортирана низа

4. Закључак

У овом раду, изнео сам основне идеје алгоритамске технике *подели па владај*. Решавајући наведене примере, схватио сам да је начин размишљања кључан при решавању задатака овог типа. Наивно приступање проблему не доноси никакве резултате, нити пружа простора за напредак. Илустрација пар проблема изнетих у овом раду подстакла ме је да не многе, непрограмарске задатке, гледам на начин који техника *подели па владај* налаже. Ефикасност њених алгоритама и размишљање „ван кутије“ не само да представљају изванредне вежбе за мозак, већ заиста праве велику разлику између искусних и неискусних програмера.

Наравно, као и свака стратегија, *подели па владај* такође има своје мане. Некада се проблеми могу решити на једноставнији а ефикаснији начин. Некада превише размипљања о „подели“ кораку, одвраћа пажњу од праве „владавине“. Податке, као и људе, некада и не треба делити. Циљ нам је некада само на дохват руке, схватање са каквим проблемом се суочавамо представља прави задатак.

5. Литература

https://military-history.fandom.com/wiki/Divide_and_rule

https://www.languagehumanities.org/what-is-a-divide-and-conquer-strategy.htm

https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

https://www.baeldung.com/cs/divide-and-conquer-strategy

https://igotanoffer.com/blogs/tech/divide-and-conquer-interview-questions

https://medium.com/techie-delight/divide-and-conquer-interview-questions-and-practice-problems-8855e45f4200

https://www.techiedelight.com/merge-sort/

https://medium.com/@choudharyarpit99/leetcode-704-binary-search-fc36f1fd60f0

Датум предаје матурског рада:

_____

Комисија:

Председник _____

Испитивач _____

Члан _____

Коментар:

Датум одбране: _____

Оцена_____ (___)