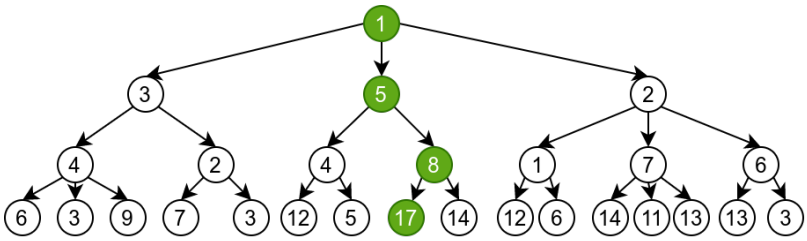
Глава 9

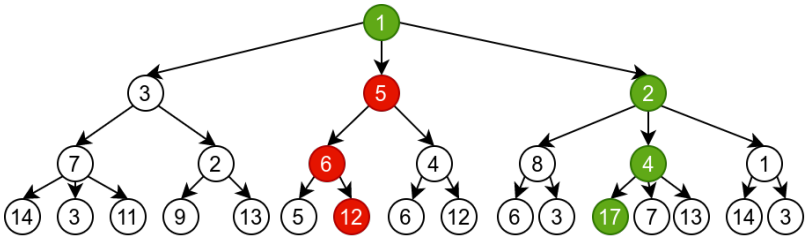
Грамзиви алгоритми

Алгоритми засновани на претрази до решења долазе кроз низ корака где у сваком кораку анализирају неколико могућности. Алгоритми код којих се уместо анализе различитих могућности у сваком кораку узима неко *локално оптимално* решење називају се **похлепни** или **грамзиви алгоритми** (енгл. greedy algorithms). Такве алгоритме обично има смисла примењивати само код проблема код којих постоји гаранција да ће такви избори на крају довести до *глобално оптималног* решења.

На слици су приказана два дрвета претраге таква да у случају првог дрвета похлепни алгоритам који у сваком кораку бира наследника са највећом могућом вредношћу доводи до оптималног решења (максималне могуће вредности), док у случају другог дрвета похлепни алгоритам доводи до неоптималног решења (вредности која није оптимална).



Слика 9.1: Дрво претраге код којег грамзиви алгоритам доводи до оптималног решења (максималне вредности)



Слика 9.2: Дрво претраге код којег грамзиви алгоритам не доводи до оптималног решења (максималне вредности)

Похлепни алгоритми нам, дакле, пружају јасну стратегију (бирај што бољу од свих расположивих могућности) како да у сваком кораку изаберемо једну од више понуђених могућности тако да на крају дођемо до жељеног оптималног решења. У наставку ћемо појам грамзивих алгоритама мало проширити и разматраћемо алгоритме који у сваком кораку бирају само једну могућност на основу неке прецизно дефинисане стратегије, тако да ти избори гарантују да ће се на крају доћи до жељеног (исправног тј. оптималног) решења проблема.

Похлепни алгоритми не врше испитивање различитих случајева нити исцрпну претрагу и стога су по правилу веома ефикасни (у сваком кораку је извршено максимално могуће одсецање). Такође, обично се веома јед-

ноставно имплементирају. Са друге стране, као и код свих других алгоритама у којима се користи одсецање, потребно је унапред доказати да се похлепним алгоритмом добија коректно тј. оптимално решење, што у неким случајевима може бити веома изазовно. Само налажење исправног похлепног алгоритма (тј. стратегије) може представљати озбиљан изазов и често није тривијално одредити да ли за неки проблем постоји или не постоји похлепно решење.

Код неких проблема похлепни алгоритми не доводе увек до оптималног решења, али се може доказати да ће решења која се добијају бити квалитетна и неће се пуно разликовати од оптималних, што може оправдати употребу похлепних алгоритама (јер они могу бити много ефикаснији од исцрпних алгоритама који гарантују оптималност). У том случају похлепни алгоритми су *хеуристике* (технике које не гарантују да ће увек довести до оптималног решења, али који доводе до довољно добрих решења).

Алгоритми засновани на претрази или на динамичком програмирању обично у сваком кораку разматрају више могућности (којима се добија више потпроблема) и након разматрања свих могућности бирају ону најбољу. Дакле, избор се врши тек након решавања потпроблема. За разлику од тога грамзиви алгоритми унапред знају која могућност ће водити до оптималног решења и избор врше одмах, након чега решавају само један потпроблем. У случају оптмизационих проблема и у случају грамзивих алгоритма потребно је да важи својство **оптималне подструктуре** тј. да се оптимално решење полазног проблема добија помоћу оптималног решења потпроблема.

Да би се доказала коректност похлепног алгоритма, обично је потребно доказати неколико ствари. Иако ћемо некада грамзивим алгоритмима решавати проблеме у којима се захтева да се испита да се провери да ли постоји неко решење које задовољава дате услове и да се пронађе било које такво решење, најчешће ћемо разматрати проблеме у којима се захтева да се у групи решења која задовољавају неке дате услове (исправних решења) пронађе оно оптимално (у случају када постоји више таквих оптималних решења обично је довољно да се пронађе било које).

1. Прво је потребно доказати да похлепна стратегија даје решење које је исправно тј. решење које задовољава све услове задатка.

2. Након тога је потребно доказати и да је решење добијено похлепном стратегијом оптимално. Ти докази су по правилу тежи и постоји неколико техника како се они изводе. Обично се крене од неког решења за које претпостављамо да је оптимално и које не мора бити идентично ономе које смо добили похлепном стратегијом. Оно не може бити горе од решења нађеног на основу похлепне стратегије (јер она враћа једно коректно решење, па оптимум може бити само евентуално бољи од тог решења), а потребно је доказати да не може бити боље.

    - Једна техника да се оптималност докаже је то да се покаже да се оптимално решење може мало по мало, применом трансформације појединачних корака може претворити у решење добијено на основу наше стратегије. Обично је довољно доказати да се први корак оптималног решења може заменити првим кораком који грамзива стратегија сугерише, тако да се коректност и квалитет решења тиме не нарушавају и коректност даље следи на основу индуктивног аргумента. Ову технику називаћемо **техника размене** (енгл. exchange).

    - Једна техника да се оптималност докаже је то да се докаже да је решење добијено на основу похлепне стратегије увек по неком критеријуму испред претпостављеног оптималног решења. Ову технику називаћемо **похлепно решење је увек испред** (енгл. greedy stays ahead).

    - Једна техника да се оптималност докаже је да се одреди теоријска граница вредности оптимума и да се онда докаже да похлепни алгоритам даје решење чија је вредност управо једнака оптимуму. Ову технику називаћемо **техника границе** (енгл. structural bound).

Задатак: Реч у реч прецртавањем слова

Дате су две речи записане малим словима. Написати програм којим се проверава да ли се друга реч може добити прецртавањем слова (не обавезно суседних) у првој речи.

Улаз: У првој линији стандардног улаза налази се прва реч, а у другој линији друга реч.

Излаз: У првој линији стандардног излаза приказати реч `da` ако се друга реч може добити прецртавањем неких слова прве речи, иначе приказати реч `ne`.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`anica`	`da`	`anica`	`ne`
`ana`		`cana`	

Решење

При прецртавању слова, њихов редослед се не мења што значи да речи редослед слова у другој речи мора бити исти као редослед прецртаних слова у првој речи. Свако слово друге речи мора да се јави у првој. Ако постоји неко исправно прецртавање, онда се исправно прецртавање може добити и тако што се у првој речи задржи прво појављивање првог слова друге речи (сва слова испред њега се прецртају) и остала слова друге речи потраже иза тог првог појављивања.

Пример. На пример, ако се у речи `bcababaca` тражи реч `abac`, тада се прецртају слова `bc`, задржи се прво `a` и затим се унутар дела `babaca` потражи реч `bac`.

Наиме, ако би постојало решење у којем је прво појављивање првог слова друге речи у првој речи прецртано, а задржано је неко његово касније појављивање, пошто се сва остала задржана слова друге речи у првој речи налазе иза тог каснијег појављивања првог слова, могли бисмо извршити размену и прецртати то задржано касније појављивање, а задржати прво појављивање и тако добити исправно прецртавање у којем је задржано баш прво појављивање.

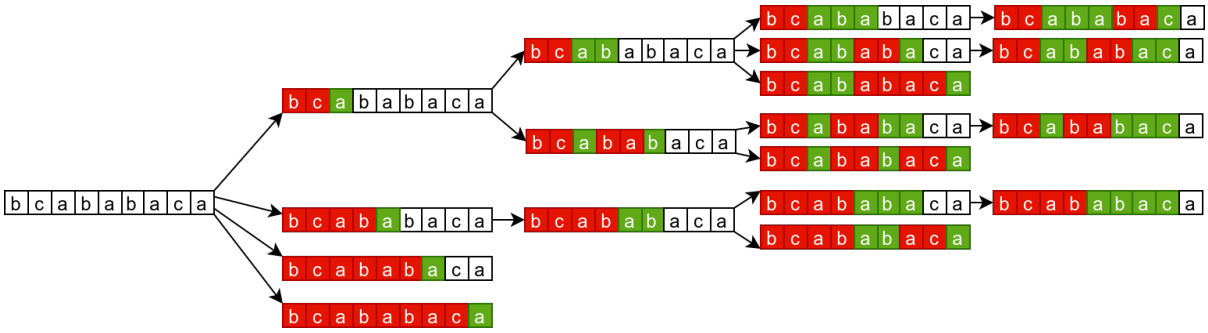
Пример. Једно могуће прецртавање слова које од речи `bcababaca` даје реч `abac` је `BCABabacA` (прецртана слова су приказана великим словима). Видимо да је прецртано прво појављивање слова `A`, док је непрецртано неко његово касније појављивање. Исправно прецртавање можемо добити тако што оставимо прво `a` које је било прецртано, а прецртамо `a` које није било прецртано. Тако добијамо прецртавање `BCaBAbacA`.

Сличан је случај и са даљим словима (увек у првој речи бирамо прво непрецртано појављивање текућег слова друге речи, јер ако реч можемо добити прецртавањем слова, можемо је добити и коришћењем те стратегије).

Пример. Размене се могу наставити. Од прецртавања `BCaBAbacA`, разменом можемо добити прецртавање `BCabABacA` и затим `BCabaBAcA`. Ово прецртавање је такво да је увек задржано прво појављивање текућег слова друге речи.

Стога се решење заснива на једноставном грамзивом алгоритму (прецртавамо прво појављивање текућег слова друге речи у првој на које наиђемо и не разматрамо остале варијанте), чија се коректност заснива на томе што се било које друго исправно решење може разменама преправити исправно решење добијено грамзивом стратегијом. Дакле, ако постоји исправно прецртавање постоји и исправно прецртавање које ће дати грамзива стратегија, што значи да ако грамзива стратегија не успе да пронађе исправно прецртавање, тада исправно прецртавање не постоји (друга реч није подниз прве).

На слици је приказано дрво исцрпне претраге приликом тражења речи `abac` унутар речи `bcababaca`. Грамзиви алгоритам избегава анализу целог дрвета тако што се креће само дуж његове горње ивице.



Нагласимо и да је наша грамзива стратегија таква да у случају када постоји више могућности, бира ону која оставља највише могућности да се преостали потпроблем успешно реши (избором првог појављивања првог слова друге речи унутар прве речи, остатак друге речи се надаље тражи у најдужем могућем преосталом делу ниске, што повећава вероватноћу да се пронађе успешно прецртавање). Ово је и општи савет за креирање грамзивих стратегија.

Довољно је пролазити у петљи кроз прву реч, карактер по карактер, и проверавати да ли је текући карактер једнак карактеру који је на реду у другој речи (на почетку, на реду је први карактер са индексом 0). Ако су одговарајући карактери једнаки, прелази се на наредни карактер у обе речи, а иначе само у првој. Петљу

треба прекинути када прођемо кроз све карактере бар једне речи. Ако смо прошли кроз све карактере друге речи, закључујемо да се друга реч може добити прецртавањем слова прве, иначе не.

Анализа сложености. Два показивача `i` и `j` се померају, први по првој, а други по другој речи само у једном смеру, па је сложеност овог алгоритма јасно $O(m+n)$, где су m и n дужине речи.

Решење можемо реализовати помоћу два бројача и петље `while`.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // redom prolazimo kroz slova obe reci dok ne dodjemo do kraja jedne
  // od njih
  int i = 0, j = 0;
  while (i < s1.size() && j < s2.size()) {
    // ako je tekuce slovo prve reci jednako tekucem slovu druge reci
    // onda ga zadrzavamo, a u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      // ako smo slovo zadrzali, prelazimo na naredno slovo druge reci
      j++;
    // prelazimo na naredno slovo prve reci
    i++;
  }

  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Други поглед на исти овај поступак је да се за сваки карактер друге речи провери да ли постоји у првој речи и ако постоји, пронађе његово прво појављивање, при чему се претрага за првим карактером врши од почетка прве речи, а за сваким наредним од позиције иза оне на којој је претходни карактер нађен. Ако се неки карактер не може пронаћи, тада другу реч није могуће добити од прве. Ако се сви карактери друге речи успешно пронађу, онда је другу реч могуће добити од прве. Претрагу карактера можемо вршити или алгоритмом линеарне претраге.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // tekuca pozicija u prvoj reci
  int i = 0, j;
  // trazimo jedno po jedno slovo druge reci u prvoj
  for (j = 0; j < s2.size(); j++) {
    // pretragu pokrecemo od pozicije i
    while (i < s1.size()) {
      // ako smo nasli slovo, prekidamo pretragu
      if (s2[j] == s1[i])
        break;
      i++;
    }
    // ako smo dosli do kraja prve reci, nismo nasli slovo i prekidamo
    // pretragu
    if (i == s1.size())
      break;
  }
  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Линеарну претрагу у делу ниске можемо вршити и библиотечким функцијама. У језику C++ може се употребити метода `find` која враћа специјалну вредност `string::npos` ако карактер није нађен. Постоји варијанта

ове методе која при карактер који се тражи и позицију од које креће претрага.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // trazimo jedno po jedno slovo druge rece u prvoj, krenuvsi od pozicije i
  int i, j;
  for (i = 0, j = 0; j < s2.size(); i++, j++) {
    // trazimo tekuce slovo drugoj reci u prvoj, krenuvsi od pozicije i
    i = s1.find(s2[j], i);
    // ako ga nismo nasli, tada prekidamo pretragu
    if (i == string::npos)
      break;
    // prelazimo na sledecu poziciju u obe reci
  }

  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Задатак: Жаба на камењу

Камење је постављено дуж позитивног дела x-осе и за сваки камен је позната његова координата x. Жаба креће да скаче са првог камена који се налази у координатном почетку и жели да у што мање скокова дође до последњег камена. У сваком скоку она може да прескочи највише растојање r (а може да скочи и мање, ако је то потребно). Написати програм који одређује да ли жаба може стићи до последњег камена и ако може у колико најмање скокова то може учинити.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредном реду n позитивних целих бројева број (у питању је растуће сортиран низ бројева који представља координате камења). У последњем реду се налази позитиван цео број r.

Излаз: На стандардни излаз исписати најмањи број скокова потребан да жаба стигне до последњег камена или -1 ако то није могуће.

Пример

Улаз	*Излаз*
5 0 3 8 14 16 10	2

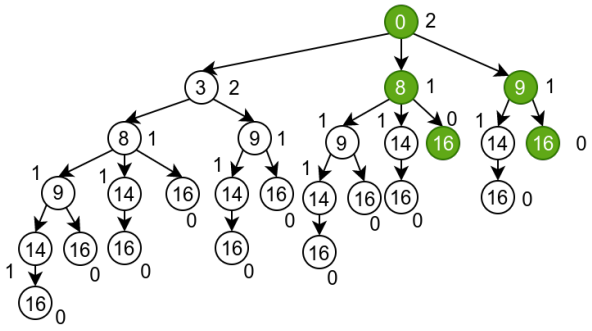
Решење

Груба сила - динамичко програмирање

Један начин да се задатак реши је грубом силом, тј. испробавањем свих могућности. Иако се на овај начин добија алгоритам чија је коректност прилично очигледна (јер су све могућности експлицитно испитане), у овом задатку такво решење је непотребно неефикасно, јер, како ћемо видети, постоји веома једноставна грамзива стратегија која увек доводи до оптималног решења (најмањег броја скокова).

Основна идеја алгоритма грубе силе је да се покуша скок са почетног камена на све каменове на које се може доскочити и да се затим рекурзивно израчуна минимални број скокова са камена на који смо доскочили до крајњег камена. Ако елиминишемо каменове на које смо доскочили са којих не постоји пут до крајњег камена и од преосталих каменова пронађемо минимални број скокова до крајењег камена, минимални број скокова од почетног камена до крајњег је број који је за један већи од тог броја.

Пример. На слици је приказано дрво исцрпне претраге за низ каменова $0, 3, 8, 9, 14, 16$ и дужину скока $r = 10$. Са камена 0, жаба може да скочи на каменове 3, 8 и 9, са камена 3 на каменове 8 и 9, са камена 8 на каменове 9, 14 и 16, са камена 9 на каменове 14 и 16 и са камена 14 на камен 16. Уз сваки чвор је обележен и најмањи број скокова потребних да жаба дође до завршног камена 16. Означене су и две најкраће путање ($0 - 8 - 16$ и $0 - 9 - 16$).



Слика 9.3: Дрво исцрпне претраге

Пошто једноставна рекурзивна имплементација доводи до поновљених рекурзивних позива (може се очекивати да ћемо током претраге често долазити на неки камен и рачунати минимални број скокова од тог камена до краја), програм треба оптимизовати динамичким програмирање.

Једно решење је да се употреби мемоизација.

Анализа сложености. Ако би се са сваког камена могло доскочити до сваког другог, овим бисмо добили решење сложености $O(n^2)$ (уз коришћење помоћног низа дужине n).

```
// minimalni broj skokova od kamena i do poslednjeg kamena n-1
int brojSkokova(const vector<int>& kamenje, int i, int r, vector<int>& memo) {
  int n = kamenje.size();

  // zaba je vec na poslednjem kamenu
  if (i == n - 1)
    return 0;

  // ako smo vec resavali ovaj problem, ocitavamo resenje
  if (memo[i] != 0)
    return memo[i];

  // minimalni broj skokova (n ima ulogu +beskonacno)
  int min = n;
  // analiziramo jedan po jedan kamen na koji zaba moze da doskoci sa kamena i
  for (int j = i+1; j < n && kamenje[j] <= kamenje[i] + r; j++) {
    // rekurzivno odredjujemo najmanji broj skokova od j do n-1
    int broj = brojSkokova(kamenje, j, r, memo);
    // ako se od j moze stici do n-1, azuriramo minimum ako je potrebno
    if (broj != n && broj + 1 < min)
      min = broj + 1;
  }
  // memorisemo i vracamo minimalni broj skokova od i do n-1
  // ako nije moguce stici, vrednost ce ostati n
  return memo[i] = min;
}

int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> memo(n, 0);
  // odredjujemo minimalni broj skokova od kamena 0 do n-1
  int broj = brojSkokova(kamenje, 0, r, memo);
  // funkcija vraca n ako nije moguce doci do n-1, a treba da vrati -1
  return broj == n ? -1 : broj;
}
```

Уместо мемоизације можемо употребити и динамичко програмирање навише. Низ у коме чувамо минимални

број скокова до крајњег камена треба попуњавати од крајњег камена налево, у опадајућем редоследу координата.

Анализа сложености. Сложеност овог решења је такође $O(n^2)$ (уз коришћење помоћног низа дужине n).

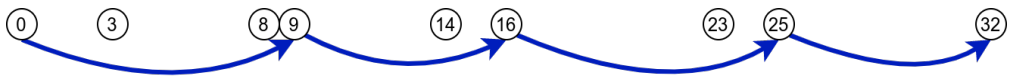
```
int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> dp(n);
  // zaba se vec nalazi na poslednjem kamenu
  dp[n-1] = 0;
  // racunamo minimalni broj skokova za svaki kamen, unazad
  for (int i = n-2; i >= 0; i--) {
    // n ima ulogu +beskonacno
    dp[i] = n;
    // analiziramo jedan po jedan kamen na koji zaba moze da
    // doskoci sa kamena i
    for (int j = i+1; j < n && kamenje[j] <= kamenje[i] + r; j++)
      // ako se od j moze stici do n-1, azuriramo minimum
      // ako je potrebno
      if (dp[j] != n && dp[j] + 1 < dp[i])
        dp[i] = dp[j] + 1;
  }

  // dp[0] je minimalni broj skokova od kamena 0 do n-1
  // dp[0] je jednako n ako nije moguce doci do n-1, a treba da vrati -1
  return dp[0] == n ? -1 : dp[0];
}
```

Оптимална стратегија - грамзиви алгоритам

Иако у сваком кораку жаба може имати избор између неколико каменова на које може да скочи, интуитивно нам је сасвим јасно да она ништа не губи тиме што скочи што даље. Стога је јасно да жаба у сваком кораку треба да скочи на што даљи камен који је на растојању највише r. Ако ниједан камен који је на растојању највише r не постоји, тада жаба не може стићи до краја. Након што жаба направи први скок, поступак се понавља на исти начин, све док не стигне до краја или док се не дође до ситуације у којој жаба не може да скочи даље.

У сваком кораку тражимо најдаљи камен на који жаба може да скочи и то тако што тражимо први камен на који жаба не може да доскочи (то можемо урадити линеарном, али и бинарном претрагом). Када нађемо камен на који жаба не може да доскочи жаба скаче на камен испред њега (ако жаба може да скочи на сваки камен, тада скаче на последњи, а ако не може да скочи ни на један камен, тада констатујемо да до последњег камена не може да доскочи).

Пример. Размотримо пример када је низ камења $0, 3, 8, 9, 14, 16, 23, 25, 32$ и када је дужина скока $r = 10$. Са почетног камена 0 жаба скаче на камен 9 (то је најдаљи камен на који може да доскочи са камена 0). Након тога скаче на камен 16 (то је најдаљи камен на који може да доскочи са камена 9), након тога на камен 25 (то је најдаљи камен на који може да доскочи са камена 16) и на крају на камен 32 (то је најдаљи камен на који може да доскочи са камена 25).



Слика 9.4: Решење добијено грамзивом стратегијом

Овај алгоритам је типичан похлепни (грамзиви) алгоритам, јер се у сваком кораку узима што је више могуће и тако да се долази и до глобалног оптимума (најмањег броја скокова). Остаје питање како доказати да је ова стратегија коректна.

Доказ коректности. Прво доказујемо да претходни алгоритам увек даје коректно решење тј. решење у складу са условом задатка (жаба креће са првог камена, долази на последњи и у сваком кораку скаче само

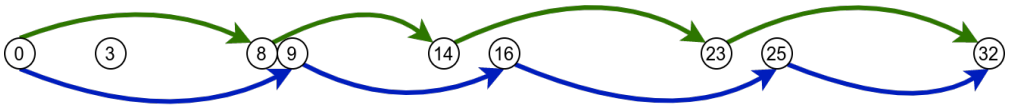
напред, не више од r метара). Заиста, имплементација је одређена тако да је сваки скок који жаба прави у претходном алгоритму скок на камен који је удаљен највише r (то се у алгоритму експлицитно проверава), па смо сигурни да је низ скокова, ако се пронађе, исправан.

Додатно је потребно да докажемо да је и у случају када функција враћа -1, тај одговор коректан тј. да тада заиста није могуће да жаба дође до краја. То се дешава када постоје два камена са координатама x_i и x_{i+1} такви да је $x_{i+1} > x_i + r$. Ако би постојало неко исправно решење, жаба би морала да скочи са неког камена пре i (или са камена i) на неки камен након x_{i+1} (или на камен x_{i+1}), но то је немогуће јер је због сортираности низа растојање сваког таквог пара каменова строго веће од r.

Друго што је потребно доказати је да је решење које се добија стратегијом оптимално. У овом случају то значи да похлепна стратегија даје најмањи могући број скокова.

Доказаћемо да се камен на ком се жаба налази након i корака примене стратегије никада не налази строго иза камена на ком се жаба налази након i корака у оптималном решењу (жаба скаче по стратегији је у сваком кораку или испред или на истом камену у односу на све жабе које достижу циљ у најмањем броју корака). Ово је типичан доказ техником *похлепно решење је увек испред*.

Пример. Размотримо пример када је низ камења $0, 3, 8, 9, 14, 16, 23, 25, 32$ и када је дужина скока $r = 10$. Решење на основу стратегије је $0, 9, 16, 25, 32$. Једно другачије оптимално решење је $0, 8, 14, 23, 32$. Заиста, ни у једном кораку решење на основу стратегије не заостаје за оптималним решењем (а у многим корацима предњачи). Важи $0 = 0$, $9 \geq 8$, $16 \geq 14$, $25 \geq 23$ и $32 = 32$.



Слика 9.5: Решење добијено грамзивом стратегијом приказано доле, предњачи у односу на било које друго оптимално решење приказано горе

Формално, нека је оптимална вредност броја скокова k и и нека је $x_0^*, x_1^*, \ldots, x_{k-1}^*$ сортиран низ x-координата камења на које жаба стаје у једном таквом оптималном решењу. Нека је $x_0, x_1, \ldots, x_{m-1}$ решење добијено на основу наше стратегије. Пошто ниједно решење не може бити боље од оптималног, важи да је $k \leq m$. Индукцијом доказујемо да за свако $i < k$ важи $x_i \geq x_i^*$.

- Базу чини случај $i = 0$ и тада је $x_0 = x_0^*$, јер се жаба у оба случаја налази на почетном камену.
- Под претпоставком да важи $x_i \geq x_i^*$ доказујемо да важи $x_{i+1} \geq x_{i+1}^*$. Ако је $x_i \geq x_{i+1}^*$ (тј. ако је у i-том кораку грамзивом стратегијом жаба већ на камену на који долази у кораку $i + 1$ у оптималном решењу или негде испред тог камена), пошто жаба скаче само напред, важи да је $x_{i+1} > x_i \geq x_{i+1}^*$. У супротном је $x_i < x_{i+1}^*$ (након i корака грамзиве стратегије жаба се налази иза камена на ком се налази након $i + 1$ корака у отпималном решењу). Пошто је оптимално решење коректно, знамо да је $x_{i+1}^* \leq x_i^* + r$. Пошто на основу индуктивне хипотезе знамо да важи $x_i \geq x_i^*$ важи и $x_{i+1}^* \leq x_i + r$. Дакле, жаба са камена x_i може сигурно да доскочи на камен x_{i+1}^* (он се налази испред камена x_i, на растојању мањем од r), а можда може и даље. Пошто грамзива стратегија узима увек најдаљи скок, важи да је $x_{i+1} \geq x_{i+1}^*$.

На основу доказаног важи и да је $x_{k-1} \geq x_{k-1}^*$, међутим, пошто је x_{k-1}^* координата последњег камена, то мора бити и x_{k-1} (па је $m = k$). Дакле, и оптимална стратегија стиже до последњег камена у k корака, па оптимално решење није боље од похлепног.

```
int brojSkokova(const vector<int>& kamenje, int r) {
  // strategija: u svakom koraku skoci sto dalje

  int n = kamenje.size();
  int broj = 0;   // broj skokova
  int kamen = 0;  // kamen na kom se zaba trenutno nalazi
  while (kamen < n - 1) {
    // odredjujemo najdalji kamen na koji mozemo stici od tekuceg
    int noviKamen = kamen;
    // dok god mozes da skocis dalje, skoci dalje
    while (noviKamen + 1 < n &&
```

```
          kamenje[noviKamen + 1] - kamenje[kamen] <= r)
      noviKamen++;
    // zaba ne moze da skoci ni na jedan kamen
    if (noviKamen == kamen)
      return -1;
    // zaba skace na najdalji moguci kamen i uvecavamo broj skokova
    kamen = noviKamen;
    broj++;
  }
  // vracamo broj skokova dobijen ovom strategijom
  return broj;
}
```

Анализа сложености. У имплементацији се користе два показивача (`kamen` и `novi_kamen`), који се кроз низ каменова крећу само унапред, тако да је укупан број корака алгоритма сигурно ограничен двоструком дужином низа и сложеност овог решења је $O(n)$.

Задатак: Шаховске екипе

Шаховска екипа A је позвала на припреме шаховску екипу B. Свака екипа има исти број такмичара и за сваког такмичара је познат рејтинг. Екипа домаћина има могућност да одабере парове који ће играти у првом колу. Ако сваки играч домаћина побеђује госта који има мањи или једнак рејтинг, а губи од госта који има строго већи рејтинг, напиши програм који одређује који је највећи број победа које екипа домаћина (екипа A) може да оствари у првом колу.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредом реду рејтинзи играча екипе домаћина (природни бројеви) раздвојени размацима, а у наредом реду рејтинзи играча екипе гостију (природни бројеви) раздвојени размацима.

Излаз: На стандардни излаз исписати само један број који представља највећи могући број победа домаћина.

Пример

Улаз	*Излаз*

`4`
`2120 1985 2205 1842`
`2045 2100 1990 1980`
	`3`

Објашњење

Домаћин може да оствари највише три победе. На пример, играч 2205 може да добије играча 2100, играч 2120 може да добије играча 2045, а играч 1985 може да добије играча 1980.

Решење

Задатак можемо решити помоћу неколико различитих грамзивих алгоритама. Преформулишимо мало задатак ради једноставније дискусије. Циљ је да одредимо највећи број парова домаћих и гостујућих играча у којима домаћи играч нема мањи рејтинг од свог пара. Зато ћемо приликом распоређивања обраћати пажњу само на оне партије у којима домаћини побеђују, док ћемо остале партије занемарити (потпуно нам је небитно како су тачно распоређени играчи у партијама у којима домаћини губе).

Најбољи домаћин против најбољег госта којег може да победи

Једна могућност је да се парови формирају тако што се упари k најбољих домаћих играча са k најлошијих гостујућих играча (док остале играче упаримо на произвољан начин). Та стратегија би била коректна, али њена имплементација није тривијална, јер није јасно колико максимално може да буде k тако да у свих k парова домаћини добијају.

Слична стратегија, која се једноставно имплементира је следећа. Ако домаћин може да оствари бар једну победу, њу може да донесе најбољи домаћи играч. Наиме, ако би он изгубио свој меч, увек бисмо неког од играча који је добио меч могли заменити њиме (јер је он бољи од свих играча домаћина) и добити исти број победа. Поставља се питање са којим гостујућем играчем он треба да игра. Циљ нам је да након формирања тог пара преостану што лошији гостујући играчи, да би слабији играчи тима домаћина имали шансе да остваре

победе. Јасно је да у скуп парова у којима домаћини добијају не можемо да укључимо госте који су бољи од тог најбољег домаћег играча (јер њих нико од играча домаћина не може да победи). Добра стратегија је да од преосталих гостујућих играча изаберемо најбољег. Након упаривања најбољег домаћег играча и госта са којим ће он да игра и елиминисања свих гостију бољих од њега, проблем је сведен на проблем истог облика, али мање димензије (смањен је број преосталих домаћих и број гостујућих играча које покушавамо да упаримо тако да домаћини побеђују). Излаз из овог рекурзивног поступка представља случај када су сви домаћи играчи успешно упарени или када међу преосталим гостујућим играчима нема лошијих од најбољег међу преосталим домаћинима.

Доказ коректности. Докажимо и формално коректност овакве грамзиве стратегије.

Решење које претходни алгоритам даје је коректно упаривање и задовољава услове задатка јер се сваки домаћин упарује са гостом која није бољи од њега (то се експлицитно проверава) и није ни упарен ни са једним другим домаћином (јер се након упаривања и домаћин и гост елиминишу из даљег разматрања). Дакле, сигурни смо да заиста постоји коректно упаривање у коме домаћин остварују број пријављен број победа.

Покажимо и да наша стратегија прави оптимални број победа за домаћу екипу. Доказ ће ићи техником размене, тј. тиме што ћемо се показати да се оптимално упаривање може трансформисати у оно добијено грамзивом стратегијом, одржавајући укупан број парова у којима домаћин побеђује (за играче домаћина који побеђују рећи ћемо да су *добитно упарени*). Посматрајмо неко оптимално упаривање. Нека је d_i најбољи домаћин који учествује у њему и нека је g_i гост са којим је он упарен.

Ако он није укупно најбољи домаћин d_s, тада најбољи домаћин сигурно није добитно упарен. Можемо домаћина d_i избацити из добитног упаривања и њему придруженог госта g_i придружити укупно најбољем домаћину d_s (то је могуће јер је $d_s \geq d_i \geq g_i$). Такво упаривање је и даље оптимално (јер се број добитних парова за домаћина није променио).

Нека је g_s гост која би био одабран стратегијом (најбољи гост који није бољи од d_s, тј. најбољи гост за кога важи $d_s \geq g_s$).

- Ако он није део тренутног добитног упаривања, онда госта g_i који тренутно игра са домаћином d_s можемо избацити и заменити гостом g_s (то је могуће јер је $d_s \geq g_s$).
- Ако јесте распоређен тако да губи од неког домаћина d_j, онда можемо направити размену тако да d_s игра са g_s, а d_j са g_i. Докажимо да је ово и даље коректно упаривање. Важи да је $d_s \geq g_s$ и $d_s \geq g_i$. Пошто је g_s најбољи гост кога d_s може да победи, важи да је $g_s \geq g_i$. Зато је $d_j \geq g_s \geq g_i$. Са ове две евентуалне размене добијамо и даље оптималан распоред који је у складу са нашом стратегијом што се тиче првог пара.

Настављајући размене по истом принципу (тј. на основу индуктивног аргумента), упаривање можемо трансформисати у оно формирано нашом стратегијом, задржавајући све време оптималност.

Приликом имплементације, скупове домаћих и гостујућих играча можемо чувати у низовима уређеним у нерастућем редоследу рејтинга и алгоритам можемо реализовати техником два показивача (продискутоваћемо касније и остале могуће варијанте). Низ домаћина обилазимо редом, елемент по елемент, а низ гостију раздвајамо на оне које су елиминисани (оне који су до сада упарени и оне које нису упарени, али су бољи од текућег домаћег играча) и преостале. Одржавамо место почетка низа гостију који још нису обрађени и приликом тражења пара за текућег домаћег играча низ гостију обилазимо од те позиције. Сваког госта или елиминишемо, јер је бољи од текућег домаћег играча или га упарујемо са текућим домаћим играчем и онда их обојицу елиминишемо. Нагласимо да се у имплементацији не морамо враћати на елиминисане госте, јер ако је неки гост бољи од текућег домаћина (најбољег међу преосталим), биће бољи и од свих наредних (преосталих) домаћина.

Анализа сложености. Пошто се оба показивача крећу само у једном смеру и сложеност фазе упаривања је линеарна. Укупним алгоритмом, дакле, доминира сложеност сортирања, па је укупна сложеност $O(n \log n)$.

Могуће је формулисати и грамзиву стратегију која ће бити дуална овој управо описаној. У тој грамзивој стратегији обрађујемо госте у неопадајућем редоследу рејтинга (од лошијих ка бољима) и сваком госту додељујемо најлошијег домаћина који може да га победи. Аналогно претходној, једноставно се доказује да је и та грамзива стратегија такође коректна.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // Ako domacini mogu da ostvare nekih k pobeda, onda tih k pobeda moze
  // da ostvari njihovih k najjacih igraca. Zato domacine obradjujemo u
```

```
  // nerastucem redosledu rejtinga i za svakog redom odredjujemo gosta kojeg
  // moze da pobedi. Za svakog domacina odredjujemo najboljeg gosta kojeg
  // moze da pobedi jer time losijim igracima domace ekipe ostavljamo prostor
  // da pobede nekoga.

  // broj igraca
  int n = domaci.size();
  // zelimo da i domacine i goste obilazimo u nerastucem redosledu rejtinga
  // (od boljih ka losijim)
  sort(begin(domaci), end(domaci), greater<int>());
  sort(begin(gosti), end(gosti), greater<int>());
  // broj pobeda domacih igraca
  int brojPobeda = 0;
  // tekuci indeks domaceg i gostujuceg igraca
  int d = 0, g = 0;
  while (true) {
    // trazimo najboljeg gosta kojeg moze da pobedi domacin na poziciji d
    // goste koji su jaci od trenutno najaceg domacina ne moze da pobedi niko
    // od preostalih domacina, pa ih eliminisemo iz daljeg razmatranja
    while (g < n && domaci[d] < gosti[g])
      g++;
    // ako najaci nerasporedjeni domacin ne moze da pobedi nijednog
    // gosta, ne mozemo povecati broj pobeda
    if (g >= n) break;
    // u suprotnom smo nasli gosta g kojeg uparujemo sa domacinom d
    brojPobeda++;
    // obojicu eliminisemo iz daljeg uparivanja
    g++, d++;
  }
  return brojPobeda;
}
```

Неефикасна имплементација

Скренимо пажњу и на важност ефикасне имплементације. Размотримо решење засновано на дуалној стратегији у којој упарујемо лоше госте. Као што смо видели, у ефикасној имплементацији би приликом преласка на сваког новог госта домаћина требало тражити само међу онима који у ранијим корацима нису елиминисани (било тако што су упарени или тако што је установљено да не могу да победе неког од слабијих гостију). Ако претрагу домаћина сваки пут почињемо из почетка (водећи рачуна о томе да раније упарене домаћине не упарујемо поново, тако што у посебном низу региструјемо оне домаћине које смо већ упарили) добићемо неефикасан алгоритам.

Анализа сложености. Пошто се за сваког од n гостију изнова претражује низ од n домаћина, сложеност најгорег случаја ове имплементације је $O(n^2)$. Нагласимо да није проблем у грамзивој стратегији, већ у њеној лошој имплементацији.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
  int n = domaci.size();
  // Ako je moguce pobediti nekih k gostiju, onda tih k gostiju mogu
  // biti k najlosijih gostiju. Zato goste obradjujemo u rastucem
  // redosledu rejtinga (od losijih ka boljima) i za svakog redom
  // odredjujemo domacina koji moze da pobedi tog gosta, a nije ranije
  // uparen.
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // broj pobeda domacih igraca
  int brojPobeda = 0;
  // da li je domacin trenutno uparen
  vector<bool> zauzet(n, false);
```

```
  // obilazimo sve goste
  for (int g = 0; g < n; g++)
    // trazimo najslabijeg neuparenog domacina koji moze pobediti gosta g
    for (int d = 0; d < n; d++)
      if (!zauzet[d] && domaci[d] >= gosti[g]) {
        brojPobeda++;
        zauzet[d] = true;
        break;
      }

  return brojPobeda;
}
```

Распоређивање сваког домаћина са најлошијим или најбољим гостом

Још једна варијанта наше победничке стратегије (тј. њене дуалне варијанте) обилази све домаћине и госте у неопадајућем редоследу рејтинга и ако домаћин може да победи најслабијег тренутно нераспоређеног госта, онда га упарујемо са њим, а у супротном га упарујемо са најјачим гостом (јер он не може победити никога од преосталих гостију, а мора бити упарен са неким, па је најбоље упарити га са најјачим гостом за ког је сада извесно да нико не може да га победи). На тај начин не добијамо само број победа, већ и ефективно упаривање свих играча у ком се постиже тај максимални број победа.

Ту стратегију можемо имплементирати тако што чувамо скуп преосталих домаћина и гостију, проналазимо најмањи тј. највећи елемент у скупу и уклањамо их. Ако се скуп имплементира преко низа (вектора, листе), добијамо веома неефикасан алгоритам.

Анализа сложености. Овај алгоритам је сложености $O(n^2)$ (јер се и проналажење минимума и максимума и уклањање елемента са дате позиције врши у линеарној сложености).

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj pobeda domacina
  int brojPobeda = 0;

  // sve dok ima preostalih domacina
  while (domaci.size() > 0) {
    // odredjujemo i uklanjamo najlosijeg domacina
    int najlosijidom = *min_element(begin(domaci), end(domaci));
    domaci.erase(find(begin(domaci), end(domaci), najlosijidom));

    // odredjujemo najlosijeg gosta
    int najlosijigost = *min_element(begin(gosti), end(gosti));
    if (najlosijidom >= najlosijigost) {
      // uparujemo najlosijeg domacina i najlosijeg gosta
      // domacin je bolji i pobedjuje
      brojPobeda++;
      // uklanjamo najlosijeg gosta
      gosti.erase(find(begin(gosti), end(gosti), najlosijigost));
    } else {
      // uparujemo najlosijeg domacina sa najboljim gostom
      int najboljigost = *max_element(begin(gosti), end(gosti));
      // uklanjamo najboljeg gosta
      gosti.erase(find(begin(gosti), end(gosti), najboljigost));
    }
  }

  return brojPobeda;
}
```

Програм постаје много ефикаснији ако употребимо библиотечке колекције које нам пружају ефикаснију имплементацију скупа (која дозвољава ефикасно тражење и уклањање минималног и максималног елемента). У језику C++ можемо употребити мултискупове (јер можда постоји више играча са истим рејтингом) које на

располагању имамо кроз колекцију `multiset`. Пошто је мултискуп уређен итератор `begin()` указује на најмањи елемент, а итератор `prev(end())` на највећи елемент. Уклањање елемента можемо извршити помоћу метода `erase`.

Анализа сложености. Под претпоставком да се операције са мултискупом врше у логаритамској сложености у односу број елемената у мултискупу, овај алгоритам ће бити сложености $O(n \log n)$.

```
int maksBrojPobeda(multiset<int>& domaci, multiset<int>& gosti) {
  int brojPobeda = 0;

  // sve dok ne rasporedimo sve domace igrace
  while (domaci.size() > 0) {
    // rasporedjujemo najboljeg domacina
    int najmanjidom = *domaci.begin();
    domaci.erase(domaci.begin());
    // sa najlosijim gostom ako moze da ga pobedi ili sa najboljim
    // gostom ako ne moze
    int najmanjigost = *gosti.begin();
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      gosti.erase(gosti.begin());
    } else {
      gosti.erase(prev(gosti.end()));
    }
  }

  return brojPobeda;
}
```

Ипак најефикасније решење добијамо ако мултискупове представимо сортираним низом, а брисање не вршимо ефективно, већ само чувамо показивач на тренутно необрађеног домаћина, док у скупу гостију необрађене госте чувамо између два показивача.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
  int n = domaci.size();
  // broj pobeda domacina
  int brojPobeda = 0;
  // sortiramo oba tima u neopadajucem redosledu rejtinga (od najslabijih do najboljih)
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // pozicija najlosijeg i najboljeeg nerasporedjenog gosta
  // svi gosti iz intervala [0, gLos) i (gDobar, n) su vec upareni
  int gLos = 0, gDobar = n-1;
  // rasporedjujemo sve domace igrace
  for (int d = 0; d < n; d++) {
    // rasporedjujemo najboljeg domacina sa najlosijim gostom ako moze
    // da ga pobedi ili sa najboljim gostom ako ne moze
    if (domaci[d] >= gosti[gLos]) {
      // uparujemo domacina d i gosta gLos
      brojPobeda++;
      gLos++;
    } else {
      // uparujemo domacina d i gosta gDobar
      gDobar--;
    }
  }
  return brojPobeda;
}
```

Задатак: Разломљени ранац

У једној продавници се продају слаткиши (бомбонице, чоколадице, кексићи) "на меру". Постоји n врста слаткиша и знамо да i-тог слаткиша има w_i грама, по укупној цени од v_i динара. Продавница је у оквиру своје промоције организовала наградну игру у којој је наградила једну своју муштерију тако да на поклон може да узме све слаткише који стају у ранац носивости W грама. Написати програм који одређује највећу вредност слаткиша које сретни добитник може да узме.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). У n наредних редова се налазе по два цела броја w_i и v_i (цели бројеви између 1 и 100). У последњем реду налази се носивост ранца W (цео број између 1 и 10^9).

Излаз: На стандардни излаз исписати највећу вредност коју срећни добитник може да покупи, заокружену на две децимале.

Пример 1

Улаз	*Излаз*
3 10 60 20 100 30 120 50	240.00

Објашњење

Максимална вредност се постиже ако се узме 10 килограма слатикша чија је укупна цена 60 динара, 20 килограма слаткиша чија је укупна цена 100 динара и 20 килограма слаткиша чија је укупна цена 120 динара (пошто се узима две трећине масе, на њему се добија вредност 80 динара).

Пример 2

Улаз

```
3
10 60
20 100
30 120
80
```

Излаз

```
280.00
```

Решење

Пошто сретни добитник жели да покупи што већу вредност слаткиша, јасно је да треба да крене узимање оних слаткиша који су највреднији тј. чија је цена по граму највећа. Ако том врстом слаткиша може да испуни цео ранац, најбоље му је да то да уради (на овом месту претпостављамо да је могуће да не покупи целокупну расположиву количину тог слаткиша). У супротном, покупиће целокупну количину тог слаткиша, а затим ће преосталу носивост ранца попунити преосталим слаткишима, по истом принципу (овде се може уочити индуктивно-рекурзивна конструкција).

Јасно је да претходни грамзиви алгоритам даје увек коректно решење, јер се ни за један предмет не узима већа количина од доступне и укупна маса узетих слаткиша не превазилази масу ранца.

Дужни смо још да докажемо да наша грамзива стратегија доводи до оптималног решења.

Доказ коректности. Користићемо метод размене. Претпоставимо да смо сортирали слаткише тако да важи

$$\frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq \ldots \geq \frac{v_{n-1}}{w_{n-1}}.$$

Свако решење је одређено узетом масом (у грамима) сваког од слаткиша. Нека је решење на основу стратегије одређено низом маса $(s_0, s_1, \ldots, s_{n-1})$, при чему за свако $0 \leq i < n$ важи $0 \leq s_i \leq w_i$ и $\sum_{i=0}^{n-1} s_i \leq W$, где је W укупна носивост ранца. Претпоставимо да је оптимално решење одређено низом маса $(o_0, o_1, \ldots, o_{n-1})$, при чему за свако $0 \leq i < n$ важи $o_i \leq w_i$ и $\sum_{i=0}^{n-1} o_i \leq W$.

Пошто је o оптимално решење, мора да важи да је $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$. Наиме, знамо да ће када се слаткиши узимају на основу стратегије укупна маса узетих слаткиша били или једнака носивости ранца или укупној расположивој маси свих слаткиша (када је она већа или једнака од носивости ранца). Ако би се у оптималном решењу садржала мања маса слаткиша од тога, могли бисмо масу неког слаткиша да повећамо и тако да добијемо још боље решење, што је у контрадикцији са претпоставком оптималности.

Претпоставимо да стратегија даје решење s које је различито од оптималног решења o. Нека је j прва позиција на којој се низови s и o разликују (за свако $0 \leq i < j$ важи да је $s_i = o_i$). Грамзива стратегија узима редом целокупне расположиве масе свих предмета, све до последњег узетог предмета где се узима максимална маса која стаје у ранац, тако да је једина могућност да је $o_j < s_j$ и то за $j < n - 1$ (због услова $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$). Размотримо решење o' које добијамо тако што уместо масе o_j предмета j узмемо масу s_j (то сигурно можемо, јер је $s_j \leq w_j$). Тиме смо повећали укупну масу у ранцу за $s_j - o_j$ и зато укупну масу узетих предмета након позиције j морамо да смањимо за $s_j - o_j$ (то можемо јер је $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$ и $\sum_{i=0}^{j-1} o = \sum_{i=0}^{j-1} s_i$). Докажимо да ово решење не може бити лошије од оптималног. То је интуитивно јасно јер смо повећали масу скупљег предмета, на рачун смањења масе масе јефтинијих предмета, но покажимо то и формално. Вредност у ранцу се повећала за $(s_j - o_j) \cdot \frac{v_j}{w_j}$ и умањила за $\sum_{k=j+1}^{n-1} s_k \frac{v_k}{w_k}$, где је $\sum_{k=j+1}^{n-1} s_k = s_j - o_j$. Докажимо да повећање вредности не може бити мање него смањење вредности тј. да се овом променом укупна вредност у ранцу није смањила. Важи да је

$$\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k} \leq \sum_{k=j+1}^{n-1} c_k \frac{v_{j+1}}{w_{j+1}} = \frac{v_{j+1}}{w_{j+1}} \sum_{k=j+1}^{n-1} c_k = \frac{v_{j+1}}{w_{j+1}}(s_j - o_j) \leq \frac{v_j}{w_j}(s_j - o_j).$$

Овом променом смо, дакле, направили решење o' чија је вредност једнака оптималној (јер ниједно решење не може бити боље од оптималног решења o, а управо смо доказали да решење o' није лошије од њега), а које се поклапа са стратегијским решењем s на једној позицији више него полазно оптимално решење o. Настављајући поступак на исти начин добићемо решење које је оптимално и једнако је s.

Анализа сложености. Сложеношћу доминира фаза сортирања предмета по јединичној цени, па је сложеност целог алгоритма $O(n \log n)$. Након сортирања, узимање предмета и израчунавање максималне вредности врши се у сложености $O(n)$.

```
double razlomljeniRanac(const vector<int>& cena,
                        const vector<int>& kolicina,
                        int nosivostRanca) {
  // broj vrsta slatkiša
  int n = cena.size();
  // vektor u kome čuvamo jedinične cene i količine svih slatkiša
  vector<pair<double, int>> jedinicnaCenaIKolicina(n);
  for (int i = 0; i < n; i++) {
    // jedinična cena slatkiša slatkiša broj i
    double jedinicnaCena = (double)cena[i] / (double)kolicina[i];
    jedinicnaCenaIKolicina[i] =
      make_pair(jedinicnaCena, kolicina[i]);
  }
  // sortiramo opadajuće na osnovu jedinične cene
  sort(begin(jedinicnaCenaIKolicina), end(jedinicnaCenaIKolicina),
       greater<pair<double, int>>());
  // ukupna vrednost slatkiša koja se može poneti u rancu
  double ukupnaVrednost = 0.0;
  // obrađujemo slatkiše po opadajućoj vrednosti jedinične cene
  for (int i = 0; nosivostRanca > 0 && i < n; i++) {
    // čitamo jediničnu cenu i količinu slatkiša broj i
    double jedinicnaCena = jedinicnaCenaIKolicina[i].first;
    int kolicina = jedinicnaCenaIKolicina[i].second;
    // uzimamo što više, ali smo ograničeni raspoloživom količinom
    // i preostalom nosivišću ranca
    int uzetaKolicina = min(kolicina, nosivostRanca);
    // preostala nosivost ranca
```

```
    nosivostRanca -= uzetaKolicina;
    // ažuriramo ukupnu vrednost
    ukupnaVrednost += uzetaKolicina * jedinicnaCena;
  }
  // vraćamo ukupnu vrednost uzetih slatkiša
  return ukupnaVrednost;
}
```

Задатак: Распоред активности

У једном кабинету се суботом одржава обука програмирања. Сваки наставник је написао термин у ком жели да држи наставу (познат је сат и минут почетка и сат и минут завршетка часа). Одреди како је могуће направити распоред часова тако да што више наставника буде укључено.

Улаз: Са стандардног улаза се учитава прво број n (укупан број наставника, $1 \leq n \leq 50000$), а затим у n наредних редова по четири броја раздвојена размацима који представљају сат и минут почетка тј. завршетка часа (претпоставити да је завршетак увек иза почетка).

Излаз: На стандардни излаз исписати највећи број наставника који могу да одрже своје часове.

Пример

Улаз	*Излаз*
7 8 15 9 20 10 45 11 30 11 20 12 45 9 30 12 40 10 20 11 20 12 00 13 00 11 30 13 30	3

Објашњење

Могу се одржати, на пример, часови од 8:15 до 9:20, затим час од 10:20 до 11:20 и на крају од 11:30 до 13:30.

Решење

Сваки час можемо представити паром бројева који представљају број минута од претходне поноћи до почетка и до краја часа (већ приликом учитавања сате и минуте можемо превести само у минуте).

Исцрпна претрага

Наивно решење се заснива на испитивању свих могућих подскупова скупа часова који су такви да се сви часови могу одржати (никоја два часа из тог скупа се не секу). Пресек два интервала постоји ако и само ако је каснији почетак часа после ранијег краја часа. Генерисање свих подскупова можемо вршити рекурзивно. Тај поступак је пописан у задатку Сви подскупови.

Анализа сложености. С обзиром на велики број подскупова које треба испитати ово решење је веома неефикасно (сложеност му је експоненцијална $O(2^n)$, где је n укупан број часова).

Грамзиви приступ

Ефикасно решење проблема се може добити грамзивим приступом. Постоји неколико грамзивих стратегија које је логично размотрити, међутим, неке од њих неће гарантовати оптималност пронађеног решења.

Један приступ може бити онај у коме прво распоређујемо час који први почиње. На слици је приказан контра-пример, који показује да се већ са три часа на тај начин може добити распоред који није оптималан.

Један приступ може бити онај у коме тежимо да распоредимо часове који кратко трају, са идејом да на тај начин остављамо више простора да се у слободним терминима одрже други часови. На слици је приказан контра-пример који показује да се већ са три часа на тај начин може добити распоред који није оптималан.

Слика 9.6: Пример на коме стратегија која распоређује час који први почиње и пример на коме стратегија која распоређује час који је најкраћи даје неоптимално решење

Једна грамзива стратегија која даје оптимално решење је следећа. Од свих нераспоређених часова бирамо онај који се најраније завршава и који се може одржати (не сече се са до сада одржаним часовима, тј. почиње након завршетка претходног часа). Интуитивно, таквим избором остављамо што већу могућност за распоређивање накнадних часова. Овим добијамо једну рекурзивну конструкцију.

- Ако је скуп часова празан, нема часова који се могу распоредити.
- У супротном бирамо час који се најраније завршава, одбацујемо часове који се са њим секу и рекурзивном правимо распоред за преостале часове.

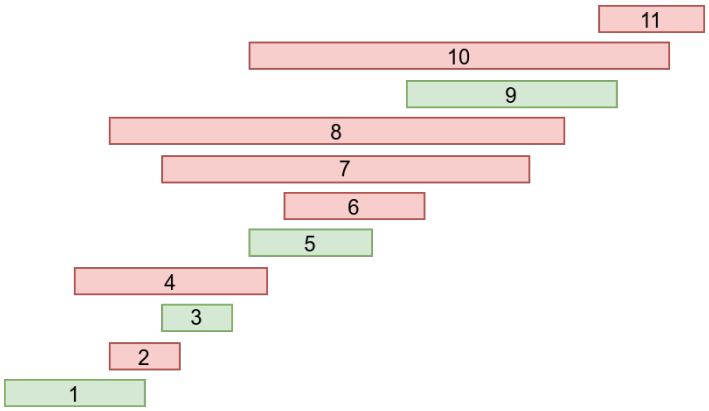
Доказ коректности. Докажимо да је ова рекурзивна формулација коректна, тако што ћемо да докажемо да увек постоји исправан, оптималан распоред (распоред са највише часова) у ком учествује час c_0 који се први завршава. Претпоставимо да је O неки исправан, оптималан распоред. Ако у њему учествује час c_0, онда је O је тај тражени распоред. Ако не учествује, онда претпоставимо да је o_0 час у O који се први завршава. Сви други часови у O почињу након завршетка часа o_0. Заиста, пошто је O исправан, ниједан час у o_i се не сече са o_0, ако би неки почињао пре o_0, тада би морао и да се заврши пре o_0, што је немогуће, јер се од свих часова у O час o_0 први завршава. Час c_0 се не завршава касније него час o_0, јер је он час који се први завршава од свих часова. Дакле, час c_0 се не сече ни са једним часом у O (осим евентуално са o_0). Када заменимо c_0 и o_0, добијамо исправан, оптималан распоред (број часова се није променио) који садржи c_0.

Дакле, јасно је да ће нас избор часа c_0 водити до неког оптималног решења. Такође је јасно да у том решењу не може да учествује ниједан час који се сече са c_0. Остатак часова бирамо рекурзивно, па коректност алгоритма следи на основу индуктивног аргумента (претпостављамо да рекурзивни позив коректно проналази оптимални распоред у преосталом скупу часова).

Техника која је употребљена у претходном доказу назива се техника замене тј. размене, јер се од неког произвољног оптималног распореда заменом добио распоред који наша грамзива стратегија бира.

Алгоритам се може формулисати и итеративно. Часове можемо сортирати неопадајуће на основу времена њиховог завршетка и обилазити их у том редоследу. Први час сигурно бирамо да буде одржан. Редом пролазимо кроз наредне часове и ако текући час почиње након последњег одабраног часа, бирамо га, а у супротном га прескачемо.

Пример. На слици су приказани часови сортирани по времену завршетка. Грамзивом стратегијом се прво одржава час 1, затим се час 2 прескаче (јер се преклапа са 1), па се затим час 3 одржава (јер се не преклапа са 1, јер је десно од њега), па се час 4 прескаче (јер се преклапа са 3), па се час 5 одржава (јер се не преклапа са 3, па самим тим ни са 1, јер је десно од њих), па се часови 6, 7 и 8 прескачу (јер се преклапају са часом 5), па се час 9 одржава (јер се не преклапа са 5, па самим тим ни са 3 ни са 1, јер је десно од њих) и на крају се прескачу часови 10 и 11 (јер се преклапају са часом 9). Максималан број часова који се могу одржати без преклапања је 4, међутим, часови 1, 3, 5 и 9 нису једино решење. Могуће је, на пример, одржати часове 1, 3, 6 и 11.



Слика 9.7: Резултат примене грамзиве стратегије

Доказ коректности. Испишимо и доказ коректности итеративне варијанте алгоритма.

Формално, претпоставимо да је $O = [o_1, o_2, \ldots, o_k]$, низ часова који представља неко исправно оптимално решење и докажимо да он садржи исти број часова као и распоред који би одабрала наша стратегија. Претпоставимо да су часови o_1 до o_k сортирани неопадајуће по редоследу њиховог завршетка. Пошто се сви ти часови могу одржати, између њих нема преклапања и сваки наредни почиње након завршетка претходног.

Нека је $S = [s_1, s_2, \ldots, s_{k'}]$ низ часова који би био одабран нашом грамзивом стратегијом. Пошто је O оптималан, S не може да садржи више часова од њега важи да је $k \leq k'$.

Докажимо прво да постоји оптималан распоред такав да за свако $1 \leq i \leq k'$ важи да је $o_i = s_i$. Тај распоред можемо добити поступним изменама почетног распореда O. Претпоставимо да постоји неко $1 \leq i \leq k'$ тако да је $o_i \neq s_i$ (у супротном је полазни распоред тај тражени). Нека је i први такав индекс тј. нека за свако $1 \leq j < i$ важи да је $o_j = s_j$. Покажимо да се заменом часа o_i часом s_i у низу O добија такође распоред који је исправан (он је свакако оптималан јер се број часова не мења). Покажимо прво да се s_i не завршава касније него o_i.

- Заиста, ако је $i = 0$, тада наша стратегија бира s_0 који се први завршава он не може да се завршава касније него o_0.
- Ако је $i > 0$, тада знамо да се o_i мора да почиње после $o_{i-1} = s_{i-1}$, међутим, наша стратегија за s_i бира онај час који почиње након s_{i-1} који се први завршава, па се s_i ни у овом случају не може завршавати касније него o_i.

Ако постоје часови у O пре часа o_i, они остају непромењени и час s_i се не преклапа са њима (јер га је стратегија бира тако да почиње након завршетка часа $s_{i-1} = o_{i-1}$). Пошто се s_i не завршава касније него o_i он се сигурно не преклапа ни са једним часом из O који иде после o_i (јер сви они почињу и завршавају се након краја часа o_i). Дакле, s_i се не преклапа ни са једним часом из O (осим евентуално са o_i, који је у склопу размене уклоњен) и распоред добијен заменом је исправан.

Наставком овог процеса замена стићи ћемо до жељеног оптималног распореда O таквог да за свако $1 \leq i \leq k'$ важи $o_i = s_i$.

Докажимо сада да није могуће да важи да је $k' < k$. Ако би важило да је $k' < k$, тада би важило да час $o_{k'+1}$ припада а не припада S. Пошто је O исправан, то би био час који би почињао после завршетка часа $o_{k'} = s_{k'}$. Међутим, није могуће да такав час постоји, јер би он почињао и завршавао се после часа s'_k, што значи да би наша грамзива стратегија морала да га одабере, што је у контрадикцији са тим да се он не налази у S. Дакле, важи да је $k' \geq k$.

Пошто је $k' \geq k$ и $k' \leq k$, важи да је $k' = k$, па наша грамзива стратегија бира исти број часова који је у неком (па и сваком) оптималном распореду. Дакле, стратегијом се добија један исправан, оптималан распоред.

Пример. Илуструјмо технику размене на којој лежи претходни доказ, тако што ћемо објаснити како да се од распореда $1, 3, 6, 11$ који је оптималан, али није у складу са нашом стратегијом добије распоред $1, 3, 5, 9$ који јесте у складу са нашом стратегијом.

- Први час где се распореди разликују је час 5 тј. 6. Заменом часа 6, часом 5 добија се распоред $1, 3, 5, 11$, који је такође исправан. Заиста, час 6 се не сече ни са часовима 1 и 3, ни са часом 11, јер је распоред

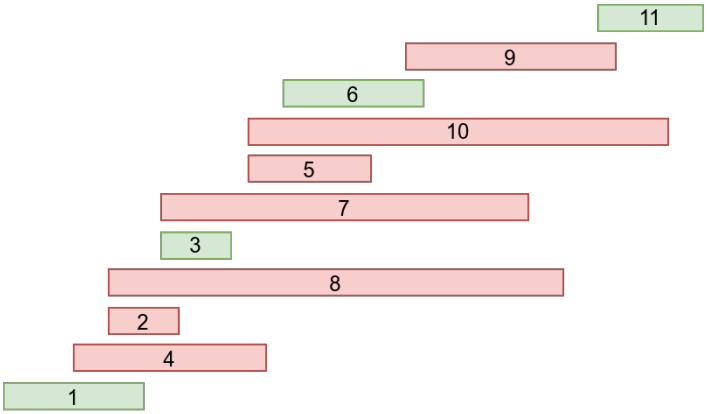
$1, 3, 6, 11$ исправан. Час 5 се не сече са часовима 1 и 3, јер га у супротном грамзива стратегија не би одабрала. Међутим, час 5 се не може завршавати после часа 6, јер се час 6 не сече са часовима 1 и 3, а од свих часова који се не секу са часовима 1 и 3, стратегија бира онај који се најраније завршава. Дакле, 5 се не завршава касније од 6, па пошто се 6 не сече са 11 и пошто је 11 потпуно десно у односу на 6, ни 5 се не сече са 11. Дакле, распоред $1, 3, 5, 11$ је исправан.

- У наредном кораку се упоређују распореди $1, 3, 5, 11$ и $1, 3, 5, 9$ (који наша стратегија препоручује). Први час где се распореди разликују је час 9 тј. 11. Заменом часа 11, часом 9 добија се распоред $1, 3, 5, 9$, који стратегија препоручује.

Дакле, произвољни оптимални распоред смо разменама трансформисали у распоред који препоручује наша стратегија, што значи да је број часова које наша стратегија распореди за одржавање оптималан.

Рецимо да постоји и дуално решење у којем се бира онај час који последњи почиње и часови се обилазе уназад, по нерастућем редоследу њиховог почетка.

Пример. На слици су приказани часови сортирани по редоследу почетка. Прво се одржава час 11, који последњи почиње, затим се прескаче час 9 који се са њим преклапа, затим се одржава час 6, након чега се прескачу часови 10, 5 и 7 који се са њим преклапају, затим се одржава час 3, прескачу се часови 8, 2 и 4 који се са њим преклапају и на крају се одржава час 1.



Слика 9.8: Резултат примене дуалне грамзиве стратегије

```
// casove predstavljamo uredjenim parovima (pocetak, kraj), u minutima
typedef pair<int, int> cas;

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

// ucitava se niz casova
vector<cas> ucitajCasove() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
```

```
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  return casovi;
}

// maksimalni broj casova koji se mogu odrzati tako da nema
// preklapanja medju rasporedjenim casovima
int maksBrojCasova(vector<cas>& casovi) {
  // broj casova
  int n = casovi.size();

  // sortiramo casove na osnovu vremena zavrsetka
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
         return krajCasa(a) < krajCasa(b);
       });

  // broj odrzanih casova
  int brojOdrzanihCasova;
  // kraj poslednjeg odrzanog casa
  int kraj;

  // rasporedjujemo prvi cas
  brojOdrzanihCasova = 1;
  kraj = krajCasa(casovi[0]);

  // analiziramo ostale casove u redosledu zavrsetka
  for (int i = 1; i < n; i++)
    // ako se tekuci cas ne preklapa sa poslednjim rasporedjenim
    if (pocetakCasa(casovi[i]) >= kraj) {
      // on se odrzava
      brojOdrzanihCasova++;
      kraj = krajCasa(casovi[i]);
    }

  return brojOdrzanihCasova;
}
```

Задатак: Распоред са најмањим бројем учионица - опис решења

За сваки од n часова познато је време почетка и завршетка. Напиши програм који одређује минимални број учионица потребан да се сви часови одрже.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у n наредних редова подаци о n часова (сат и минут почетка и сат и минут завршетка, при чему је време почетка строго мање од времена завршетка).

Излаз: На стандардни излаз исписати тражени број потребних учионица.

Пример

Улаз	*Излаз*
6 8 0 8 45 10 0 10 45 9 0 9 45 8 30 9 15 10 45 11 30 10 30 11 15	2

Објашњење

Један могући распоред наведих 6 часова у две учионице је следећи:

```
Учионица 1          Учионица 2

8:00-8:45 Час 1     8:30-9:15 Час 4
9:00-9:45 Час 3     10:30-11:15 Час 6
10:00-10:45 Час 2
10:45-11:30 Час 5
```

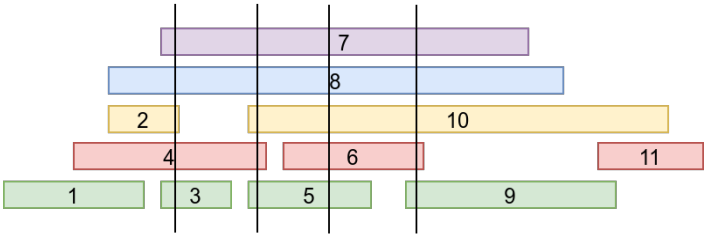
Решење

Иако донекле делује слично задатку Распоред активности, решење овог задатка захтева другачију стратегију.

Пошто се захтева да се сви часови одрже, обилазићемо их у одређеном редоследу и сваки час ћемо придруживати некој од слободних учионица. У тренутку у ком почиње неки час и у коме нема више слободних учионица које су раније отворене, отвараћемо нову учионицу у коју ћемо распоређивати тај час.

Знамо да је најмањи број учионица сигурно већи или једнак највећем броју часова који се истовремено одржавају у неком тренутку. У наставку ћемо доказати да је минималан број учионица увек једнак том броју и да се распоред може направити ако се часови обилазе у растућем редоследу њиховог почетка.

Пример. На слици је приказан један могући распоред часова. Учионице се налазе једна изнад друге (сви часови у истој учионици су приказани на истој висини). Вертикалним линијама су означени тренуци у којима су све учионице попуњене тј. тренуци у којима постоји 5 часова који се преклапају. Због тога је јасно да није могуће направити распоред са мање од 5 учионица.

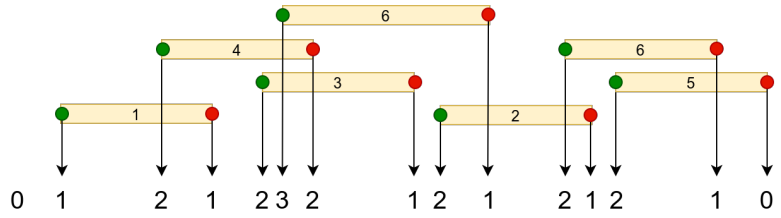


Слика 9.9: Распоред часова

Доказ коректности. Докажимо да је наша стратегија оптимална. Стратегија је таква да је једини разлог да се нова учионица отвори то да су све раније отворене учионице већ попуњене у тренутку када почиње текући час, тј. да постоји неки час (рецимо $[s_j, f_j)$) који се преклапа са свим часовима који су распоређени и у тренутку његовог почетка се одржавају у тренутних d отворених учионица. Пошто су часови сортирани на основу времена почетка, свих тих d часова почиње пре тренутка s_j и завршава се након тренутка s_j (јер трају у тренутку s_j). То значи да у тренутку s_j сигурно постоји $d+1$ часова (тих d већ распоређених и час $[s_j, f_j)$) који се у том тренутку одржавају, па број учионица мора бити бар $d+1$. Дакле, ако се на основу стратегије резервише нова учионица, сигурни смо да је то неопходно. Због тога знамо да када наша стратегија направи распоред са неким бројем учионица, сигурни смо да није било могуће направити распоред са мањим бројем учионица, што значи да је направљени распоред оптималан.

Приметимо да је у овом доказу одређена граница квалитета решења (распоред не може бити у мање учионица него што је број часова у највећој групи часова који се одржавају у неком тренутку) и затим је показано да похлепно решење достиже ту границу, па је због тога оптимално.

Ако нас занима само број потребних учионица, а и конкретан распоред, тада можемо направити веома једноставну имплементацију која одржава број тренутно отворених учионица и обилази све значајне временске тренутке (почетке и завршетке часова) редом. На крају часа смањује се број заузетих учионица, а на почетку часа повећава се број учионица. Тражени резултат се добија као максимална вредност бројача заузетих учионица.



Слика 9.10: Обилазак свих карактеристичних тачака уз ажурирање бројача

Потребно је још обратити пажњу на специјалан случај када се неки часови завршавају у истом тренутку у ком други часови почињу. Елегатно решење је да у датом временском тренутку прво обрадимо све часове који се завршавају у том тренутку и ослободимо учионице, а затим да обрадимо часове који почињу у том тренутку (тако нове часове распоређујемо у управо ослобођене учионице, што је у складу са тим да је време трајања сваког часа полуотворени интервал $[s, f)$).

Анализа сложености. За n часова постоји $2n$ карактеристичних тренутака. Њихово сортирање се врши у времену $O(n \log n)$. Обилазак сортираног низа тренутака и ажурирање бројача се затим врши у времену $O(n)$. Сложеношћу, дакле, доминира сортирање и укупна сложеност је $O(n \log n)$.

```
struct Vreme {
  int minut;
  bool pocetak;
};

Vreme napraviVreme(int sat, int min, bool pocetak) {
  Vreme v;
  v.minut = 60*sat + min;
  v.pocetak = pocetak;
  return v;
}

// ucitava spisak casova sa standardnog ulaza
vector<Vreme> ucitajCasove() {
  int n;
  cin >> n;
  vector<Vreme> vremena;
  vremena.reserve(2*n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    vremena.push_back(napraviVreme(pocSat, pocMin, true));
    vremena.push_back(napraviVreme(krajSat, krajMin, false));
  }
  return vremena;
}

// funkcija rasporedjuje casove u ucionice, vraca minimalni potrebni
// broj ucionica i svakom casu dodeljuje broj ucionice
int minUcionica(vector<Vreme>& vremena) {
  sort(begin(vremena), end(vremena),
       [](const Vreme& v1, const Vreme& v2) {
         return v1.minut < v2.minut || (v1.minut == v2.minut && !v1.pocetak && v2.pocetak);
       });
  // trenutan broj zauzetih ucionica
  int brojUcionica = 0;
  // maksimalni broj zauzetih ucionica u nekom trenutku
  int maksBroj = 0;
  for (const Vreme& v : vremena)
```

```
    if (v.pocetak)
      maksBroj = max(++brojUcionica, maksBroj);
    else
      brojUcionica--;
  return maksBroj;
}
```

Ако желимо да направимо конкретан распоред, имплементација је мало компликованија. Први корак у имплементацији је веома једноставан – учитавамо све часове у низ и сортирамо их на основу почетног времена. Кључни корак у другој фази је одређивање учионице у коју може бити смештен текући час. За све до тада отворене учионице знамо времена завршетка часова у њима. Можемо пронаћи учионицу у којој се час најраније завршава и проверити да ли је могуће да у њу распоредимо текући час. Ако јесте, њој ажурирамо време завршетка часа, а ако није, онда знамо да су све учионице заузете (јер се час који се први завршава још није завршио), па морамо отворити нову учионицу. Да бисмо ефикасно могли да нађемо учионицу у којој се час најраније завршава, све учионице можемо чувати у реду са приоритетом сортираном по времену завршетка часа у свакој од учионица. Ако тај ред није празан и ако је време завршетка часа у учионици на врху реда мање или једнако времену почетка текућег часа, време завршетка часа у тој учионици ажурирамо на време завршетка текућег часа (најлакше тако што ту учионицу избацимо из реда и поново је додамо са ажурираним временом). У супротном у ред додајемо нову учионицу којој је време завршетка часа постављено на време завршетка текућег часа (број учионице је за један већи од дотадашњег броја учионица у реду).

Број отворених учионица је увек једнак броју елемената реда. Наиме, елемент у ред додајемо само када отварамо нову учионицу јер су све до тада отворене учионице у реду заузете, док у супротном само мењамо елемент који је био на врху реда новим (часови који су се завршили а нису замењени новим часовима у истој учионици остају у реду).

Анализа сложености. Укупна сложеност алгоритма је $O(n \log n)$ – и у фази сортирања и у фази распоређивања (јер се читање и избацивање минимума, као и убацивање новог елемента у ред са приоритетом врши у времену $O(\log n)$). Када бисмо уместо реда са приоритетом користили обичан низ и у њему стално тражили минимум, сложеност најгорег случаја би порасла на $O(n^2)$.

```
struct Cas {
  // redni broj casa (njegov jedinstveni identifikator)
  int broj;
  // minut pocetka i kraja casa
  int pocetak, kraj;
  // redni broj ucionice u kojoj se cas odrzava
  int ucionica;
};

Cas napraviCas(int broj, int pocSat, int pocMin, int krajSat, int krajMin) {
  Cas c;
  c.broj = broj;
  c.pocetak = pocSat*60 + pocMin;
  c.kraj = krajSat*60 + krajMin;
  return c;
}

struct Ucionica {
  // broj ucionice (njen jedinstveni identifikator)
  int broj;
  // minut od kog je ucionica slobodna
  int slobodnaOd;
};

Ucionica napraviUcionicu(int slobodnaOd, int broj) {
  Ucionica u;
  u.broj = broj;
  u.slobodnaOd = slobodnaOd;
  return u;
```

```
}

// ucitava spisak casova sa standardnog ulaza
vector<Cas> ucitajCasove() {
  int n;
  cin >> n;
  vector<Cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(i, pocSat, pocMin, krajSat, krajMin);
  }
  return casovi;
}

// funkcija rasporedjuje casove u ucionice, vraca minimalni potrebni
// broj ucionica i svakom casu dodeljuje broj ucionice
int minUcionica(vector<Cas>& casovi) {
  // sortiramo časove na osnovu vremena njihovog početka
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.pocetak < c2.pocetak;
       });

  // red sa prioritetom u kom su trenutno zauzete ucionce slozene po
  // vremenu zavrsetka casa
  struct PorediUcionice {
    bool operator()(const Ucionica& u1, const Ucionica& u2) {
      return u1.slobodnaOd > u2.slobodnaOd;
    }
  };
  priority_queue<Ucionica, vector<Ucionica>, PorediUcionice> redUcionica;
  for (Cas& c : casovi) {
    int brojUcionice;
    if (redUcionica.empty() || redUcionica.top().slobodnaOd > c.pocetak)
      brojUcionice = redUcionica.size() + 1;
    else {
      brojUcionice = redUcionica.top().broj;
      redUcionica.pop();
    }
    c.ucionica = brojUcionice;
    redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
  }

  // sortiramo časove na osnovu rednog broja
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.broj < c2.broj;
       });

  return redUcionica.size();
}
```

Задатак: Мали поштар

Јовица зарађује џепарац тако што доноси пакете својим комшијама. Поделу креће од своје куће и потребно је да пакете разнесе у друге куће распоређене дуж улице и да се врати назад у своју кућу. За сваку кућу познато је растојање од почетка улице. Најкраћи пут би прешао ако би пакете сложио тако да их редом дели

комшијама дуж улице. Пошто Јовица жели да буде у доброј физичкој форми, он током поделе пакета трчи и жели да пакете уреди тако да пређе што већи пут. Напиши програм који одређује највећи пут који може да пређе.

Улаз: Са стандардног улаза се уноси број кућа у које треба донети пакете (међу њима се налази и Јовицина кућа), а затим и растојања тих кућа од почетка улице.

Излаз: На стандардни излаз исписати највеће растојање које Јовица може прећи током поделе пакета.

Пример 1

Улаз	*Излаз*
5 7 3 6 10 2	24

Објашњење

Постоји више начина да Јовица претрчи 24 дужне јединице. На пример, ако је његова кућа на позицији 3, он може да редом обилази куће 3, 7, 2, 10, 6, 3.

Пример 2

Улаз

```
7
3 5 11 4 2 17 9
```

Излаз

```
56
```

Решење

Груба сила

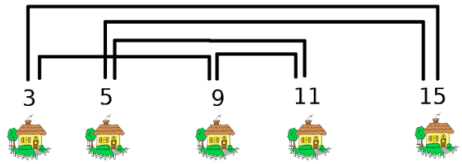
Решење грубом силом подразумева да се провере сви могући редоследи обиласка n кућа, тј. свих $n!$ пермутација датих бројева и да се утврди која од њих даје највећу могућу вредност пређеног пута. Пермутације се могу обићи коришћењем техника приказаних у задатку Све пермутације.

Анализа сложености. Алгоритам заснован на провери свих пермутација је изразито неефикасан, његова сложеност је $O(n!)$ и може се применити само на веома, веома мале улазе (практично, само за $n \leq 10$ програм може да реши задатак у датом временском ограничењу).

```
int zbirApsRazlika(const vector<int>& a) {
  int zbir = 0;
  for (size_t k = 1; k < a.size(); k++)
    zbir += abs(a[k] - a[k-1]);
  zbir += abs(a[a.size()-1] - a[0]);
  return zbir;
}

int najduziPut(vector<int>& a) {
  int maks = 0;
  do {
    maks = max(zbirApsRazlika(a), maks);
  } while(next_permutation(begin(a), end(a)));
  return maks;
}
```

Грамзива стратегија

Интуитивно нам је јасно да ће се пуно претрчати ако се стално трчи са једног на други крај улице. Један грамзиви приступ је да се прво обиђе крајња лева кућа, па затим крајња десна, па друга слева, па друга здесна и тако "цик-цак".



Слика 9.11: Цик-цак обилазак: креће се из 3, затим се иде у 15, па у 5, па у 11, па у 3 и на крају назад у 3

Доказ коректности. Докажимо да је ово грамзиво решење исправно.

За дати распоред $x_0, \ldots x_{n-1}$ одређујемо суму апсолутних вредности разлика елемената тј. покушавамо да максимизујемо суму

$$|x_0 - x_1| + |x_1 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|.$$

Сваки елемент се у суми јавља тачно два пута. У зависности од међусобног односа бројева неки елементи ће бити узети са знаком $+$, а неки са знаком $-$, и то тако да се тачно n елемената узима са знаком $+$ и тачно n елемената узима са знаком $-$. Циљ нам је да елементи који се узимају са знаком $+$ буду што већи, а да ови са знаком $-$ буду што мањи. Распоред који иде цик-цак постиже да се за знаком $+$ узме $\frac{n}{2}$ већих елемената низа, а са знаком $-$ узме $\frac{n}{2}$ мањих елемената низа (ако их је непаран број, тада се средњи узима једном са знаком $-$, а једном са знаком $+$). Заиста, ако, на пример, имамо 6 елемената $a_0 \leq a_1 \leq a_2 \leq a_3 \leq a_4 \leq a_5$, цик-цак распоред даје вредност

$$|a_0 - a_5| + |a_5 - a_1| + |a_1 - a_4| + |a_4 - a_2| + |a_2 - a_3| + |a_3 - a_0|,$$

што је једнако

$$(a_5 - a_0) + (a_5 - a_1) + (a_4 - a_1) + (a_4 - a_2) + (a_3 - a_2) + (a_3 - a_0),$$

тј.

$$2 \cdot (a_5 + a_4 + a_3) - 2 \cdot (a_2 + a_1 + a_0)$$

Јасно је да се не може добити више од овога, а ово се, видели смо, увек може експлицитно достићи баш цик-цак распоредом.

У овом примеру смо коректност грамзиве стратегије доказали тако што смо израчунали теоријско ограничење функције која се оптимизује и затимо смо доказали да се грамзивом стратегијом достиже то теоријско ограничење.

Елементе низа можемо експлицитно сортирати, па затим распоредити елементе у нови низ по цик-цак редоследу и за тај нови низ израчунати збир апсолутних вредности разлика суседних елемената.

Анализа сложености. Сложеношћу алгоритма доминира фаза сортирања, чија је сложеност $O(n \log n)$. Распоређивање елемената у помоћни низ и израчунавање дужне врши се у сложености $O(n)$.

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));
  vector<int> b(n);
  int i = 0, j = n-1;
  for (int k = 0; k < n; k++)
    if (k % 2 != 0)
      b[k] = a[i++];
    else
      b[k] = a[j--];
```

```
  return zbirApsRazlika(b);
}
```

Помоћни низ се може веома једноставно избећи у имплементацији.

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));
  int i = 0, j = n-1;
  int zbir = 0;
  bool paran = true;
  while (i < j) {
    zbir += abs(a[j]-a[i]);
    if (paran)
      i++;
    else
      j--;
    paran = !paran;
  }
  zbir += abs(a[0] - a[i]);
  return zbir;
}
```

Ако пажљиво размотримо доказ коректности, примећујемо да распоред у коме идемо од прве до последње, па до друге, затим претпоследње куће итд., није једини који даје максимални пређени пут. Довољно је само да наизменично узимамо елементе из прве и друге половине низа (у било ком редоследу). Ово инспирише још једноставнији алгоритам за израчунавање траженог максимума (саберемо $\frac{n}{2}$ бројева са почетка, одузмемо $\frac{n}{2}$ бројева са краја низа и помножимо са 2).

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));

  int zbir = 0;
  for (int i = 0; i < n/2; i++) {
    zbir += a[n-1-i];
    zbir -= a[i];
  }
  return zbir * 2;
}
```

Задатак: Исплата са посебним новчићима

У Србији се користе апоени од 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000 и 5000 динара. Написати програм који формира датих износ динара од што мањег броја апоена (новчаница и новчића) и исписује употребљен број апоена.

Улаз: Са стандардног улаза се учитава један цео број између 0 и 100 000.

Излаз: На стандардни излаз написати најмањи број апоена потребних да се исплати учитани износ новца.

Пример

Улаз	*Излаз*
243	5

Објашњење

$243 = 200 + 20 + 20 + 2 + 1$.

Решење

Један директан начин да се проблем реши је да се испитају сва могућа разлагања датог износа на збирове састављене од ових бројева и да се пронађе онај збир који има најмањи број новчића (једноставности ради,

занемаримо разлику између новчића и новчаница). Решење овог типа би се могло засновати на динамичком програмирању комбинованом са одсецањем током претраге. Такво решење приказано је у задатку Исплата са најмање новчића.

Доказ коректности није тежак, али је овај приступ био прилично компликован и неефикасан. Наиме специфичности апоена омогућавају да се задатак реши много ефикасније.

```
// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(const vector<int>& v, int n, int S) {
  vector<int> dp(S+1);
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
    // minimalni broj novčića da se naplati iznos s (pretpostavljamo
    // da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
      // proveravamo da li je iznos s moguće naplatiti novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}
```

И без формалног математичког објашњења, сваки продавац у продавници и на пијаци зна да се оптимално решење добија тако што се у сваком тренутку враћа највећи апоен који је мањи или једнак од тренутног износа и након тога се исти принцип примењује на преостали износ све док се не врати цео кусур (у питању је, дакле, грамзива индуктивно-рекурзивна конструкција). Ово решење је веома ефикасно, лако се имплементира, међутим, доказ његове коректности није нимало очигледан.

Наиме, постојање новчића од 4 динара би покварило ситуацију. 8 динара би се могло добити од два новчића од 4 динара, док би грамзива стратегија употребила три новчића (од 5, 2 и 1 динар). Дакле, доказ коректности мора да укључи анализу конкретних апоена који су у оптицају и мале промене ових апоена могу да утичу на то да описани приступ даје или не даје увек оптимално решење.

Докажимо сада коректност. Једноставности ради, претпоставићемо да су у оптицају само апоени од 1, 2, 5, 10, 20 и 50 динара (за веће новчиће доказ иде по истом принципу). Методом размене доказаћемо горње границе броја новчића од свих апоена у оптималном решењу.

- Оптимално решење не може да садржи више од једног новчића од 1 динар. Када би постојала макар два новчића од 1 динар, они би могли бити замењени једним новчићем од 2 динара, чиме би се број употребљених новчића смањио, што је у контрадикцији са претпоставком да је полазно решење оптимално. Потпуно аналогно се доказује да оптимално решење не може садржати ни више од једног новчића од 10 динара.
- Даље, оптимално решење не може садржати више од два новчића од 2 динара. Наиме, ако би садржало бар три новчића од 2 динара, они би могли бити замењени једним новчићем од 1 и једним новчићем од 5 динара, чиме би се добило мање решење, што је у контрадикцији са претпоставком да је полазно решење оптимално. Потпуно аналогно, оптимално решење не може да садржи ни више од два новчића од 20 динара.
- На крају, у оптималном решењу не може бити више од једног новчића од 5 динара. Наиме, ако би постојала бар два, она би могла бити замењена једним новчићем од 10 динара чиме би се добило мање решење, што је контрадикција.
- У решењу није могуће ни да истовремено постоје два новчића од 2 динара и новчић од 1 динара, јер би се сви они могли заменити са једним новчићем од 5 динара, што је опет контрадикција. Аналогно важи и за новчиће од 10 и 20 динара.

Узевши у обзир претходна ограничења, размотримо максималне износе са оптималним бројем новчића, који се могу добити коришћењем само одређених скупова новчића. Испоставиће се да су максимални износи увек за један мањи од првог већег апоена.

- Новчићи од 1 динар могу да направе највише износ од 1 динара (јер се смеју појавити само једном).
- Новчићи од 1 и 2 динара могу да направе највише износ од 4 динара (јер може бити највише два новчића од 2 динара и у том случају се не сме користити и новчић од 1 динар).
- Новчићи од 1, 2 и 5 динара могу да направе највише износ од 9 динара (јер не може бити више од једног новчића од 5 динара, два новчића од 2 и једног новчића од 1 динара, а ако има два новчића од 2 динара, не сме се јавити и новчић од 1 динара).
- Слично, новчићи од 1, 2, 5 и 10 динара могу да направе највише износ од 19 динара (јер се 10 динара може јавити само једном, а од 1, 2 и 5 се може направити највише 9).
- Новчићи од 1, 2, 5, 10 и 20 могу да направе највише износ од 49 динара (јер 1, 2 и 5 могу да направе највише 9, како смо објаснили, а пошто се 10 динара јавља највише једном, а 20 динара највише два пута, али не сва три таква заједно, од 10 и 20 се може направити највише 40).

Докажимо сада да се за сваки позитиван износ у оптималном решењу мора налазити највећи новчић који је мањи или једнак од тог износа (све наведене констатације се односе само на оптимална решења).

- За износ 1 јавља се само новчић 1.
- Износи између 2 и 4 динара морају да садрже новчић 2. Наиме, не може да се јави новчић од 5 динара, само од новчића од 1 динар може да се направи највише 1 динара, па за износе од 2 до 4 динара мора да се јави бар један новчић од 2 динара.
- Износи између 5 и 9 динара морају да садрже новчић од 5 динара. Наиме, не могу да садрже новчић од 10 динара, а пошто се од новчића од само 1 и 2 динара може направити највише 4 динара, за износе од 5 до 9 динара мора да се употребни новчић од 5 динара.
- Износи између 10 и 19 динара морају да садрже новчић 10. Наиме, 20 динара не може да се јави, а помоћу новчића од 1, 2 и 5 динара највише се може направити 9 динара.
- Износи између 20 и 49 динара морају да садрже бар један новчић од 20 динара. Наиме, не могу да садрже 50, а само са 1, 2, 5 и 10 се може добити највише 19.
- Износи преко 50 динара морају да садрже новчић од 50 динара. Наиме, само са новчићима од 1, 2, 5, 10 и 20 је могуће направити само 49 динара.

Дакле, успели смо да за сваки износ пронађемо новчић који оптимално решење мора да садржи, чиме онда успевамо да смањимо димензију проблема и да до решења дођемо директно, без било какве претраге и испробавања разних могућности.

```
int minBrojApoena(int iznos) {
  int brojApoena = 0;
  vector<int> apoeni{5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1};
  while (iznos > 0) {
    for (int apoen : apoeni)
      if (iznos >= apoen) {
        iznos -= apoen;
        brojApoena++;
        break;
      }
  }
  return brojApoena;
}
```

Задатак: Кодирање текста са што мање битова

Потребно је кодирати текст низом битова (нула и јединица), тако да је свако слово кодирано неким бинарним кодом. При том, кодирање мора да буде такво да није могуће да кôд неког слова буде префикс кода неког

другог слова (јер тада дешифровање не би било једнозначно). Напиши програм који одређује најмањи број битова потребних да се кодира текст.

Напомена: није неопходно користити исти број битова за кодирање сваког карактера.

Улаз: Прва линија стандардног улаза садржи број n ($1 \leq n \leq 256$) различитих карактера које треба кодирати. Након тога свака наредна линија садржи карактер и његов број појављивања у тексту.

Излаз: На стандардни излаз треба исписати тражени минимални број битова.

Пример

Улаз	*Излаз*

`5`
`a 10`
`b 28`
`c 7`
`d 14`
`e 19`
	`173`

Објашњење

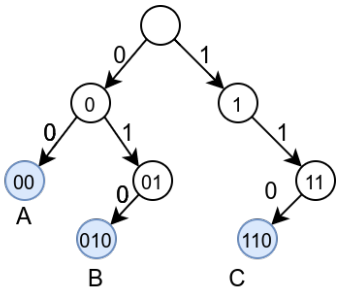
На пример, карактер `a` можемо кодирати кодом `011`, карактер `b` кодом `11`, карактер `c` кодом `010`, карактер `d` кодом `00` и карактер `e` кодом `10`.

Решење

Алгоритам који приказујемо познат је под именом *Хафманово кодирање* и представља један од класичних примера грамзивих (похлепних алгоритама).

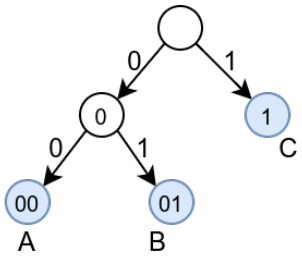
Један од услова да се обезбеди једнозначно декодирање је да ниједна кодна реч не буде префикс некој другој (такви се кодови називају префиксни кодови). Такви се кодови називају *бинарни префиксни кодови*. За дати скуп карактера и фреквенције њиховог појављивања желимо да конструишемо оптимални бинарни префиксни кôд (онај код којег је број битова $\sum_{i=0}^{n-1} f_i b_i$ потребан за запис целог текста најмањи, где је f_i број појављивања карактера c_i у тексту, а b_i број битова додељених том карактеру).

Бинарним префискним кодовима једнозначно одговарају бинарна дрвета. Кôд сваког чвора се добија обилазком дрвета – корак на лево додаје симбол 0 на код, а корак на десно симбол 1, при чему се карактерима које је потребно кодирати додељују кодови искључиво у листовима дрвета (зато није могуће да је кôд било ког карактера префикс кода неког другог карактера).



Слика 9.12: Пример бинарног префиксног кода. Слово A се кодира битовима 00, слово B битовима 010, а слово C битовима 110.

Размотримо прво структуру дрвета које одговара бинарном префиксном коду. Први важан закључак је да у оптималном префиксном коду сви унутрашњи чворови морају имати оба детета. У супротном се кôд може скратити тако што се унутрашњи чвор уклони и замени својом децом. У претходном примеру, дрво можемо трансформисати тако што унутрашњи чвор `01` заменимо својим дететом и тако карактеру `B` доделимо код `01` уместо `010`. Слично, чвор `11`, а затим и чвор `1` можемо заменити својом децом и тако карактеру `C` можемо доделити кôд `1`.



Слика 9.13: Пример бинарног префиксног кода. Слово A се кодира битовима 00, слово B битовима 01, а слово C битом 1.

Интуиција нам говори да је пожељно карактерима који се појављују често додељивати краће кодове. Зато ће карактери који се јављају ређе имати дуже кодове.

Доказ коректности. Ово није тешко формално доказати. Претпоставимо да карактер c_i има фреквенцију појављивања f_i и да се кодира са b_i битова, а да карактер c_j има фреквенцију појављивања f_j и да се кодира са b_j битова, при чему је $f_i \geq f_j$, док је $b_i \leq b_j$. Ако се замене кодови та два карактера, укупан потребан број битова за кодирање текста се не може повећати. Пошто се осталим карактерима кодови не мењају, број битова зависи само од ова два карактера. У првом случају он је једнак $f_i b_i + f_j b_j$, а у другом је једнак $f_i b_j + f_j b_i$. Ако посматрамо разлику $(f_i b_j + f_j b_i) - (f_i b_i + f_j b_j)$ добијамо израз $f_i(b_j - b_i) + f_j(b_i - b_j)$ тј. $(f_i - f_j)(b_j - b_i)$. Пошто су оба чиниоца ненегативна и полазна разлика је ненегативна, па се укупан потребан број битова овом трансформацијом није могао повећати (ако је разлика позитивна, онда се број битова смањио).

Наредно важно тврђење је то да постоји оптимално дрво у којем се два карактера који се најређе јављају, налазе као два суседна листа (сина истог оца) и то најудаљенија од корена.

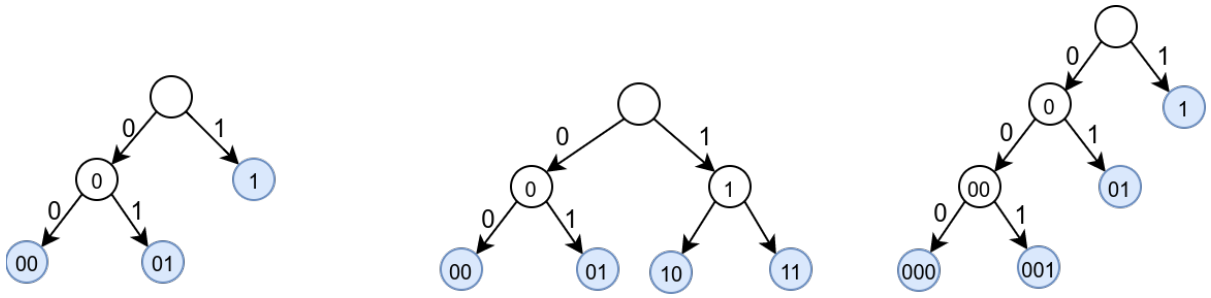
Доказ коректности. Нека су c_1 и c_2 два карактера са најмањим фреквенцијама. Размотримо неко оптимално дрво и његова два листа најудаљенија од корена. Нека су њима придружени карактери c_1' и c_2'. Можемо заменити c_1 и c_1' и заменити c_2 и c_2'. Структура дрвета остаје идентична, па се након размене и даље добија исправан бинарни префиксни кôд. Пошто су након ове размене кодови замењени тако да су чешћи карактери добили краће кодове, на основу претходно доказаног тврђења укупан број битова за кодирање текста се није могао повећати, па је добијени кôд и даље оптималан.

Пример. Покушајмо да кроз неколико примера стекнемо мало интуицију о проблему и његовом решењу.

Ако азбука има један карактер или два карактера, ситуација је потпуно јасна – ти карактери се могу кодирати са по једним битом.

Ако постоје три карактера, онда није могуће да постоје два карактера који се кодирају са по једним битом, јер би тада кôд трећег карактера садржао као префикс кôд неког од та два карактера (он би морао да почне било нулом, било јединицом). Дакле, у тој ситуацији један карактер мора бити кодиран једним битом, а друга два карактера са по два бита. Одлука који карактер треба да буде кодиран са једним битом је једноставна – то мора да буде карактер који се најчешће јавља.

Ако постоје четири карактера, тада постоји могућност да сви буду кодирани са по два бита (2-2-2-2) или да један од њих буде кодиран са једним битом, један са два бита и два са по три бита (1-2-3-3). Анализом могућих облика дрвета показује се да би сва друга кодирања била неповољнија.



Слика 9.14: Једнина могућа структура оптималног дрвета са три листа и једине две могуће структуре оптималног дрвета са четири листа, ако се занемари могућност симетричног пресликавања, тј. размене левог и десног детета

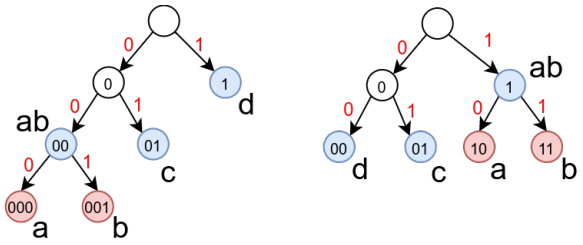
Поново је јасно да карактери који се често појављују треба да буду кодирани са мање, а они који се ретко појављују треба да буду кодирани са више битова, међутим, нејасно је од чега зависи да ли је боље употребити кодирање 2-2-2-2 или 1-2-3-3. Размотримо два примера.

```
frekvencije
a b c d
1 2 5 6

kodiranje 2-2-2-2
1·2 + 2·2 + 5·2 + 6·2 = 28

kodiranje 1-2-3-3
1·3 + 2·3 + 5·2 + 6·1 = 25
```

```
frekvencije
a b c d
3 4 5 6

kodiranje 2-2-2-2
3·2 + 4·2 + 5·2 + 6·2 = 36

kodiranje 1-2-3-3
3·3 + 4·3 + 5·2 + 6·1 = 37
```

У првом случају је повољније кодирање 1-2-3-3, а у другом 2-2-2-2. У оба случаја два најређа карактера `a` и `b` се кодирају са истим бројем битова (два или три), карактер `c` се кодира са два бита, док се најчешћи карактер `d` кодира са два или са једним битом. Кодирање 1-2-3-3 је повољније ако је $1 \cdot f_d + 3 \cdot (f_a + f_b) \leq 2 \cdot f_d + 2 \cdot (f_a + f_b)$, а кодирање 2-2-2-2 у супротном (јер се у првом случају `a` и `b` кодирају са три бита, а `d` са једним, док се у другом случају `a`, `b` и `d` кодирају са по два бита). Дакле, веома битан податак нам је збир фреквенција два карактера који се најређе јављају (збир фреквенција $f_a + f_b$ карактера `a` и `b`) и тај збир треба поредити са фреквенцијама других карактера (пошто се ти карактери увек кодирају са истим бројем битова, битан нам је само збир њихових фреквенција, а не њихов појединачан однос).

Када бисмо из оптималног дрвета са четири листа уклонили два листа која представљају најређе карактере, добили бисмо дрво оптималне структуре са три листа. Узевши у обзир симетрије тј. могућност размене левог и десног детета, оно увек има јединствени облик. Његови листови ће бити карактери `c` и `d` и трећи чвор испод којег ће бити постављени касније карактери `a` и `b` (чвор родитељ листова `a` и `b` у крајњем дрвету). Јасно је да, како год да су друга два чвора распоређена, карактеру `c` морају бити придружена два бита (ако би му се придружио 1 бит, карактеру `d` би била придружена 2, па би се њиховом разменом добио бољи код). Дакле, или ће карактеру `d` бити придружен 1 бит, а родитељу карактера `a` и `b` 2 бита или ће карактеру `d` бити придружена 2 бита, а родитељу карактера `a` и `b` 1 бит. Ако је кодирање 1-2-3-3 повољније, тада важи и да је $1 \cdot f_d + 2 \cdot (f_a + f_b) \leq 2 \cdot f_d + 1 \cdot (f_a + f_b)$ (одузимањем вредности $f_a + f_b$ у почетној неједнакости), а то је неједнакост чијом се анализом одређује како треба распоредити чвор `d` и родитељски чвор чворова `a` и `b` у дрвету са три листа — у првом случају би се карактер `d` кодирао са једним битом, а родитељски чвор, када бисмо га третирали као лист и када би му се придружио неки карактер, би се кодирао са два бита, док би се у другом случају карактер `d` кодирао са два бита, а родитељски чвор са једним битом.



Слика 9.15: Питање положаја листова a и b своди се на питање оптималног распоређивања листова c, d и "листа" ab, са фреквенцијом $f_a + f_b$

Дакле, основа Хафмановог алгоритма лежи у идеји да се уместо два карактера са најмањим фреквенцијама f_i и f_j посматра њихов родитељски чвор, као нови карактер чија је фреквенција $f_i + f_j$ и да се оптимално распоређивање тог, редукованог, скупа чворова, након чега се два најређа карактера додају испод свог родитељског чвора.

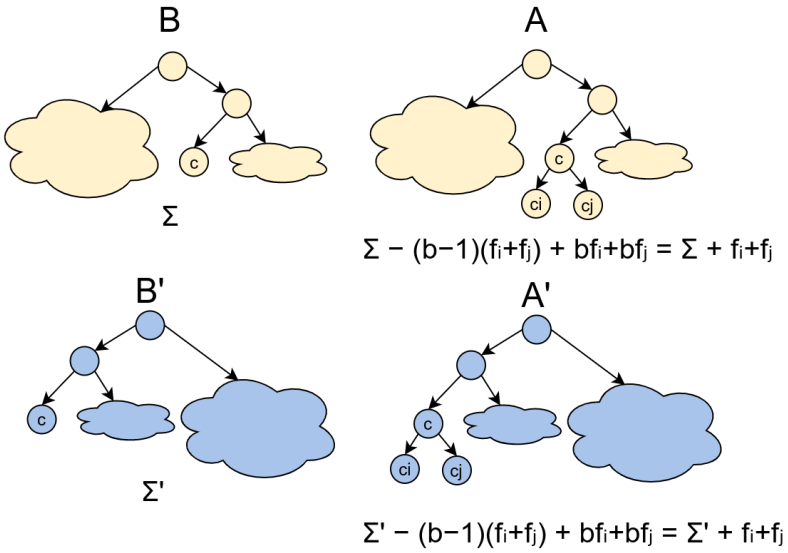
Опишимо сада Хафманову индуктивно-рекурзивну конструкцију на основу које конструишемо оптимални префиксни кôд. Индукција тече по броју карактера у азбуци за коју конструишемо кôд.

- Базу индукције чини случај када азбука има један или два карактера. Њима можемо доделити једнобитне кодове и то је сигурно оптимално.

- Редукцију на мању димензију проблема ћемо остварити тако што ћемо променити азбуку тако да два најређа карактера c_i и c_j са фреквенцијама f_i и f_j заменити новим карактером c са фреквенцијом $f_i + f_j$. Рекурзивно конструишемо дрво B (оптимални префиксни кôд) за промењену азбуку. У њему се карактер c јавља као неки лист. Дрво A (кôд) за полазну азбуку ћемо добити тако што ћемо испод листа c додати листове c_i и c_j.

Коректност поступка лежи на тврђењу да овако конструисано дрво A представља оптимални префиксни кôд за полазну азбуку.

Доказ коректности. Заиста, ако дрво A не би било оптимално за полазну азбуку, постојало би боље дрво за њу (рецимо A'). На основу претходно доказаног можемо претпоставити да се карактери c_i и c_j у њему јављају као два суседна листа (сина истог оца). Уклонимо та два листа и њиховом оцу доделимо карактер c. Тако добијамо дрво B' које одређује префиксни кôд за кодирање редуковане азбуке. Претпоставимо да кôд одређен дрветом B за кодирање текста укупно захтева Σ битова, а да кôд одређен дрветом B' за кодирање текста укупно захтева Σ' битова. Укупан број битова у дрвету A и B се разликује само за битове којима се кодирају карактери c_i, c_j и нови карактер c. У дрвету A то је $f_i b + f_j b$ битова, где је b број битова потребан за кодирање карактера c_i и c_j, док је у дрвету B то $(f_i + f_j)(b-1)$, јер је карактеру c придружена фреквенција $f_i + f_j$, док је висина чвора c за 1 мања од висине чворова c_i и c_j. Дакле, дрво A одређује кôд који за кодирање текста захтева $\Sigma - (f_i + f_j)(b-1) + (f_i b + f_j b) = \Sigma + f_i + f_j$ битова. На сличан начин се може закључити да дрво A' одређује који за кодирање текста захтева $\Sigma' + f_i + f_j$. Дакле, ако A није оптимално дрво, тада је $\Sigma' + f_i + f_j < \Sigma + f_i + f_j$, па је зато $\Sigma' < \Sigma$, што значи да ни B није оптимално дрво (јер је B' захтева мање битова за кодирање целог текста над трансформисаном азбуком). Ово је контрадикција у односу на претпоставку да смо рекурзивним позивом добили оптимално дрво B за трансформисану азбуку.



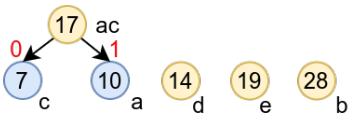
Слика 9.16: Доказ оптималности индуктивно-рекурзивне конструкције Хафмановог кода

Дакле, одређујемо два карактера са најмањим фреквенцијама појављивања, мењамо их новим карактером чија је фреквенција једнака збиру њихових фреквенција, рекурзивно конструишемо оптимално дрво и на крају у том дрвету на лист који одговара новом карактеру дописујемо два нова листа који одговарају уклоњеним карактерима са најмањим фреквенцијама. Базу индукције тј. излаз из рекурзије представља случај када остану само један или два карактера (које кодирамо са по једним битом, тј. креирамо корен дрвета испод којег се налази тај карактер тј. та два карактера).

Пример. Прикажимо рад алгоритма на примеру

```
a 10
b 28
c 7
d 14
e 19
```

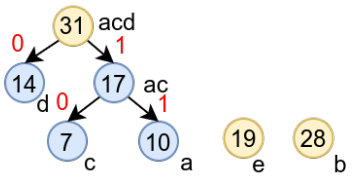
Најмањи пар фреквенција је 7 и 10 тако да крећемо од дрвета



Док азбука постаје

```
b   28
d   14
e   19
ac  17
```

Најмањи пар сада чине 14 и 17 тако да добијамо дрво

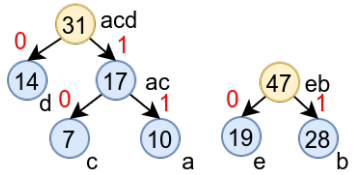


док азбука постаје

```
b 28
e 19
```

```
acd 31
```

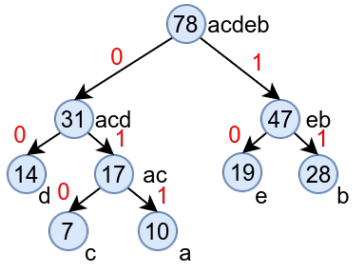
Сада је најмањи пар 19 и 28 и тако добијамо шуму



док азбука постаје

```
be  47
acd 31
```

На крају завршавамо тако што спајамо ова два карактера и добијамо дрво



Одавде читамо кодове: `a 011`, `b 11`, `c 010`, `d 00` и `e 10`.

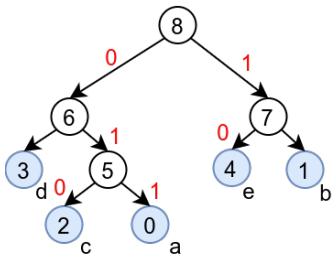
Потребно је прецизирати још неколико детаља да бисмо направили имплементацију.

У сваком кораку је у скупу карактера потребно одређивати два са најмањим фреквенцијама и мењати их са новим, чија је фреквенција једнака збиру фреквенција. Ове операције се могу веома ефикасно извршавати ако се карактери убаце у ред са приоритетом (хип) сортиран неопадајуће на основу фреквенција. У језику C++ на располагању имамо `priority_queue` који нам пружа функционалност реда са приоритетом (потребно је једино још нагласити да ред треба да буде сортиран растуће тј. да се на врху реда налази елемент са најмањом фреквенцијом (а не највећом, како је то подразумевано).

Друго питање је то како репрезентовати дрво које се гради. Унапред знамо да ће оно имати $2n-1$ чворова и знамо да ће n карактера бити листови тог дрвета, док ће $n-1$ чворова бити унутрашњи. Сваки чвор можемо нумерисати бројевима од 0 до $2n-2$. Листовима ћемо делити индексе у складу са редним бројем карактера који су им придружени c_i (карактеру c_i придружен је чвор број i). Индексе унутрашњим чворовима (бројеви од n до $2n-2$) додељиваћемо у редоследу њиховог формирања (корен ће бити формиран последњи и њему ће бити додељен индекс $2n-2$). За сваки од $n-1$ унутрашњих чворова треба да знамо индекс левог и десног детета (увек постоје оба). Те информације можемо чувати у два помоћна низа. Унутрашњи чворови имају индексе од n до $2n-2$ и за чвор број i у првом низу на позицији $i-n$ чувамо индекс левог детета, а у другом низу на позицији $i-n$ чувамо индекс десног детета.

Пример. Индекси чворова за дрво из претходног примера су приказани на слици. Низови који одређују леву и десну децу унутрашњих чворова су:

```
i      5 6 7 8
i-n    0 1 2 3
levi:  2 3 4 6
desni: 0 5 1 7
```



Слика 9.17: Индекси чворова у дрвету

У реду са приоритетом ћемо уз фреквенцију сваког чвора чувати и индекс (то ће бити позиција која одговара том чвору у низовима у којима чувамо опис деце).

Када је дрво креирано, кодове свих карактера можемо добити исцрпним обиласком целом дрвета (рекурзивном функцијом).

```
// svakom karakteru u listu drveta ciji je koren cvor sa indeksom i i kodom
// kod dodeljujemo kod i upisujemo ga u mapu kodovi
void procitajKodove(const vector<int>& levo,
                    const vector<int>& desno,
                    const vector<char>& karakteri,
                    int i, int n, const string& kod,
                    map<char, string>& kodovi) {
  if (i < n)
    // stigli smo do lista list
    kodovi[karakteri[i]] = kod;
  else {
    // u pitanju je unutrasnji cvor, pa obradjujemo levo i desno poddrvo
    procitajKodove(levo, desno, karakteri, levo[i-n], n, kod + "0", kodovi);
    procitajKodove(levo, desno, karakteri, desno[i-n], n, kod + "1", kodovi);
  }
}

// za dati niz karaktera i frekvencije njihovog pojavljivanja u tekstu
// odredjuje optimalni prefiksni kod
map<char, string> hafman(const vector<char>& karakteri, const vector<int>& frekvencije) {
  // broj karaktera
  int n = karakteri.size();

  // reprezentacija drveta - svakom unutrasnjem cvoru dodeljujemo levi i desni indeks
  vector<int> levo(n-1), desno(n-1);

  // red sa prioritetom koji sadrzi cvorove odredjene njihovim frekvencijama i indeksima
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
  // ubacujemo sve listove drveta (karaktere) u red - dodeljujemo im indekse od 0 do n-1
  for (int i = 0; i < n; i++)
    pq.emplace(frekvencije[i], i);

  // gradimo unutrasnje cvorove
  for (int i = n; i < 2*n - 1; i++) {
    // povezujemo dva cvora sa najnizim frekvencijama
    auto f1 = pq.top(); pq.pop();
    auto f2 = pq.top(); pq.pop();
    // ubacujemo cvor i u drvo
    levo[i - n] = f1.second; desno[i - n] = f2.second;
    // redukujemo azbuku i dodajemo joj kombinovani karakter koji odgovara cvoru i
    pq.emplace(f1.first + f2.first, i);
  }
```

```
  // iz drveta ocitavaom kodove svih karaktera
  map<char, string> kodovi;
  procitajKodove(levo, desno, karakteri, 2*n-2, n, "", kodovi);
  return kodovi;
}

// za dati niz karaktera i frekvencije njihovog pojavljivanja u tekstu,
// i dati prefiksni kod odredjuje broj bita potrebnih za kodiranje teksta
int brojBitova(const vector<char>& karakteri, const vector<int>& frekvencije,
               const map<char, string>& kodovi) {
  // broj karaktera
  int n = karakteri.size();

  // izracunavamo ukupan broj bitova potrebnih za kodiranje teksta
  int broj = 0;
  for (int i = 0; i < n; i++)
    broj += frekvencije[i] * kodovi.at(karakteri[i]).size();

  return broj;
}
```