3

অ্যালগরিদম বিশ্লেষণ

একবার আমরা আমাদের পরিকল্পিত একত্রীকরণ সম্পূর্ণ করার পরে কোম্পানির বেতন-ভাতা প্রক্রিয়া করতে কতক্ষণ সময় লাগবে? আমার কি বিক্রেতা X বা বিক্রেতা Y থেকে একটি নতুন বেতনের প্রোগ্রাম কেনা উচিত? যদি একটি নির্দিষ্ট প্রোগ্রাম ধীর হয়, এটি কি খারাপভাবে প্রয়োগ করা হয় বা এটি একটি কঠিন সমস্যা সমাধান করে?
এই জাতীয় প্রশ্নগুলি আমাদেরকে একটি সমস্যার অসুবিধা, বা সমস্যা সমাধানের জন্য দুই বা ততোধিক পদ্ধতির আপেক্ষিক দক্ষতা বিবেচনা করতে বলে।

এই অধ্যায়টি অ্যালগরিদম বিশ্লেষণের অনুপ্রেরণা, মৌলিক স্বরলিপি এবং মৌলিক প্রযুক্তিগত বৈশিষ্ট্যগুলি উপস্থাপন করে। আমরা অ্যাসিম্পটোটিক অ্যালগরিদম বিশ্লেষণ বা কেবল অ্যাসিম্পটোটিক বিশ্লেষণ নামে পরিচিত একটি পদ্ধতিতে ফোকাস করি । অ্যাসিম্পটোটিক বিশ্লেষণ একটি অ্যালগরিদমের সম্পদ খরচ অনুমান করার চেষ্টা করে। এটি আমাদের একই সমস্যা সমাধানের জন্য দুই বা ততোধিক অ্যালগরিদমের আপেক্ষিক খরচ তুলনা করতে দেয়। অ্যাসিম্পটোটিক বিশ্লেষণ অ্যালগরিদম ডিজাইনারদের একটি বাস্তব প্রোগ্রাম বাস্তবায়ন করার আগে একটি প্রস্তাবিত সমাধান কোনও সমস্যার জন্য সম্পদের সীমাবদ্ধতাগুলি পূরণ করতে পারে কিনা তা অনুমান করার জন্য একটি সরঞ্জাম দেয়। এই অধ্যায় পড়ার পরে, আপনি বুঝতে হবে

- একটি বৃদ্ধির হারের ধারণা, যে হারে একটি অ্যালগরিদমের খরচ বৃদ্ধি পায়
  এর ইনপুট আকার বৃদ্ধির সাথে সাথে;
- একটি বৃদ্ধির হারের জন্য উপরের এবং নিম্ন সীমার ধারণা এবং একটি সাধারণ প্রোগ্রাম, অ্যালগরিদম বা সমস্যার জন্য এই সীমাগুলি কীভাবে অনুমান করা যায়; এবং
- একটি অ্যালগরিদম (বা প্রোগ্রাম) এর খরচ এবং এর খরচের মধ্যে পার্থক্য
  একটি সমস্যা।

একটি প্রোগ্রামের খরচ পরীক্ষামূলকভাবে পরিমাপ করার সময় এবং প্রোগ্রামের দক্ষতা উন্নত করার জন্য কোড টিউনিংয়ের কিছু নীতির সম্মুখীন হওয়া ব্যবহারিক অসুবিধাগুলির একটি সংক্ষিপ্ত আলোচনার মাধ্যমে অধ্যায়টি শেষ হয়।

3.1 ভূমিকা

দক্ষতার পরিপ্রেক্ষিতে কিছু সমস্যা সমাধানের জন্য আপনি কীভাবে দুটি অ্যালগরিদম তুলনা করবেন? একটি উপায় হল উভয় অ্যালগরিদমকে কম্পিউটার প্রোগ্রাম হিসাবে প্রয়োগ করা এবং তারপর

প্রতিটি প্রোগ্রামের কতটা রিসোর্স ব্যবহার করছে তা পরিমাপ করে একটি উপযুক্ত পরিসরে ইনপুটগুলিতে চালান। এই পদ্ধতি প্রায়ই চারটি কারণে অসন্তোষজনক।
প্রথমত, প্রোগ্রামিং এবং দুটি অ্যালগরিদম পরীক্ষা করার প্রচেষ্টা জড়িত যখন আপনি সর্বোত্তমভাবে শুধুমাত্র একটি রাখতে চান। দ্বিতীয়ত, যখন অভিজ্ঞতাগতভাবে দুটি আল-গরিদম তুলনা করা হয় তখন সবসময় সুযোগ থাকে যে একটি প্রোগ্রাম অন্যটির চেয়ে "ভালো লেখা" ছিল এবং অন্তর্নিহিত অ্যালগরিদমগুলির আপেক্ষিক গুণাবলী তাদের বাস্তবায়নের দ্বারা প্রকৃতপক্ষে প্রতিনিধিত্ব করা হয় না। এটি বিশেষভাবে ঘটতে পারে যখন প্রোগ্রামারের অ্যালগরিদম সম্পর্কে পক্ষপাতিত্ব থাকে। তৃতীয়ত, অভিজ্ঞতামূলক পরীক্ষার ক্ষেত্রে পছন্দ অন্যায়ভাবে একটি অ্যালগরিদমের পক্ষে হতে পারে। চতুর্থত, আপনি দেখতে পাচ্ছেন যে দুটি অ্যালগরিদমের মধ্যেও ভালো আপনার রিসোর্স বাজেটের মধ্যে পড়ে না। সেক্ষেত্রে আপনাকে অবশ্যই একটি নতুন অ্যালগরিদম প্রয়োগ করে অন্য একটি প্রোগ্রামের সাথে পুরো প্রক্রিয়াটি আবার শুরু করতে হবে। কিন্তু, কোন অ্যালগরিদম সম্পদ বাজেট পূরণ করতে পারে কিনা তা আপনি কিভাবে জানবেন? বাজেটের মধ্যে কোনো বাস্তবায়নের জন্য সম্ভবত সমস্যাটি খুব কঠিন।

এই সমস্যাগুলি প্রায়ই অ্যাসিম্পোটিক বিশ্লেষণ ব্যবহার করে এড়ানো যায়। অ্যাসিম্প-টোটিক বিশ্লেষণ একটি অ্যালগরিদমের কার্যকারিতা পরিমাপ করে, বা একটি প্রোগ্রাম হিসাবে এটির বাস্তবায়ন, ইনপুট আকার বড় হওয়ার সাথে সাথে। এটি আসলে একটি আনুমানিক কৌশল এবং দুটি প্রোগ্রামের আপেক্ষিক যোগ্যতা সম্পর্কে আমাদের কিছু বলে না যেখানে একটি সর্বদা অন্যটির চেয়ে "সামান্য দ্রুত" হয়। যাইহোক, অ্যাসিম্পটোটিক বিশ্লেষণ কম্পিউটার বিজ্ঞানীদের জন্য উপযোগী প্রমাণিত হয়েছে যাদের অবশ্যই নির্ধারণ করতে হবে যে একটি নির্দিষ্ট অ্যালগরিদম বাস্তবায়নের জন্য বিবেচনা করা মূল্যবান কিনা।

একটি প্রোগ্রামের জন্য গুরুত্বপূর্ণ সংস্থান প্রায়শই এটির চলমান সময়। যাইহোক, আপনি একা চলার সময় মনোযোগ দিতে পারবেন না। আপনাকে অবশ্যই অন্যান্য বিষয়গুলির সাথেও উদ্বিগ্ন হতে হবে যেমন প্রোগ্রামটি চালানোর জন্য প্রয়োজনীয় স্থান (প্রধান মেমরি এবং ডিস্ক স্পেস উভয়ই)। সাধারণত আপনি একটি অ্যালগরিদমের জন্য প্রয়োজনীয় সময় (বা একটি প্রোগ্রাম আকারে একটি অ্যালগরিদমের ইনস্ট্যান্টেশন) এবং ডেটা কাঠামোর জন্য প্রয়োজনীয় স্থান বিশ্লেষণ করবেন।

অনেক কারণ একটি প্রোগ্রামের চলমান সময় প্রভাবিত করে। কিছু পরিবেশের সাথে সম্পর্কিত যেখানে প্রোগ্রামটি সংকলিত এবং চালানো হয়। এই জাতীয় কারণগুলির মধ্যে রয়েছে কম্পিউটারের CPU, বাস এবং পেরিফেরাল হার্ডওয়্যারের গতি। কম্পিউটারের সংস্থানগুলির জন্য অন্যান্য ব্যবহারকারীদের সাথে প্রতিযোগিতা একটি প্রোগ্রামকে ক্রল করতে ধীর করে দিতে পারে। প্রোগ্রাম-মিং ভাষা এবং একটি নির্দিষ্ট কম্পাইলার দ্বারা উত্পন্ন কোডের গুণমান একটি উল্লেখযোগ্য প্রভাব ফেলতে পারে। অ্যালগরিদমকে একটি প্রোগ্রামে রূপান্তরকারী প্রোগ্রামারের "কোডিং দক্ষতা" একটি দুর্দান্ত প্রভাবও ফেলতে পারে।

আপনার যদি একটি নির্দিষ্ট কম্পিউটারে সময় এবং স্থানের সীমাবদ্ধতার মধ্যে একটি প্রোগ্রাম কাজ করার প্রয়োজন হয় তবে এই সমস্ত কারণগুলি প্রাসঙ্গিক হতে পারে। তবুও, এই কারণগুলির কোনওটিই দুটি অ্যালগরিদম বা ডেটা স্ট্রাকচারের মধ্যে পার্থক্যকে সম্বোধন করে না। ন্যায্য হতে, একই সমস্যা সমাধানের জন্য দুটি অ্যালগরিদম থেকে প্রাপ্ত প্রো-গ্রাম উভয়ই হওয়া উচিত

একই কম্পাইলার দিয়ে কম্পাইল করা হয় এবং একই অবস্থায় একই কম্পিউটারে চালানো হয়। যতটা সম্ভব, বাস্তবায়নকে "সমান দক্ষ" করার জন্য প্রতিটি প্রোগ্রামে নিবেদিত প্রোগ্রামিং প্রচেষ্টায় একই পরিমাণ যত্ন নেওয়া উচিত। এই অর্থে, উপরে উল্লিখিত সমস্ত কারণের তুলনা বাতিল করা উচিত কারণ তারা উভয় অ্যালগরিদমের ক্ষেত্রে সমানভাবে প্রযোজ্য।

আপনি যদি সত্যিই একটি অ্যালগরিদমের চলমান সময় বুঝতে চান, তবে মেশিনের গতি, প্রোগ্রামিং ভাষা, কম্পাইলার এবং আরও অনেক কিছুর চেয়ে বিবেচনা করা আরও উপযুক্ত অন্যান্য কারণ রয়েছে। আদর্শভাবে আমরা স্ট্যান্ডার্ড বেঞ্চমার্ক অবস্থার অধীনে অ্যালগরিদমের চলমান সময় পরিমাপ করব। যাইহোক, কিছু কম্পিউটারে অ্যালগরিদমের বাস্তবায়ন চালানো ছাড়া আমাদের কাছে নির্ভরযোগ্যভাবে চলমান সময় গণনা করার কোন উপায় নেই। চলমান সময়ের জন্য একটি সারোগেট হিসাবে অন্য কিছু পরিমাপ ব্যবহার করা একমাত্র বিকল্প।

একটি অ্যালগরিদমের কর্মক্ষমতা অনুমান করার সময় প্রাথমিক বিবেচনার বিষয় হল একটি নির্দিষ্ট আকারের একটি ইনপুট প্রক্রিয়া করার জন্য অ্যালগরিদম দ্বারা প্রয়োজনীয় মৌলিক ক্রিয়াকলাপের সংখ্যা। "মৌলিক ক্রিয়াকলাপ" এবং "আকার" শব্দ দুটিই বরং অস্পষ্ট এবং বিশ্লেষণ করা অ্যালগরিদমের উপর নির্ভর করে। আকার প্রায়শই প্রক্রিয়াকৃত ইনপুটের সংখ্যা। উদাহরণস্বরূপ, বাছাই করার অ্যালগরিদমগুলির তুলনা করার সময়, সমস্যাটির আকার সাধারণত বাছাই করা রেকর্ডের সংখ্যা দ্বারা পরিমাপ করা হয়। একটি মৌলিক ক্রিয়াকলাপে অবশ্যই এমন বৈশিষ্ট্য থাকতে হবে যেটি সম্পূর্ণ হওয়ার সময় তার অপারেন্ডের নির্দিষ্ট মানের উপর নির্ভর করে না। দুটি পূর্ণসংখ্যা ভেরিয়েবল যোগ করা বা তুলনা করা বেশিরভাগ প্রোগ্রামিং ভাষার মৌলিক ক্রিয়াকলাপের উদাহরণ। n পূর্ণসংখ্যা সম্বলিত একটি অ্যারের বিষয়বস্তুর সংক্ষিপ্তকরণ করা হয় না, কারণ খরচ নির্ভর করে n-এর মান (অর্থাৎ, ইনপুটের আকার) উপর।

উদাহরণ 3.1 n পূর্ণসংখ্যার অ্যারেতে সবচেয়ে বড় মান খুঁজে পাওয়ার সমস্যা সমাধানের জন্য একটি সাধারণ অ্যালগরিদম বিবেচনা করুন। অ্যালগরিদম পালাক্রমে প্রতিটি পূর্ণসংখ্যা দেখে, এখন পর্যন্ত দেখা সবচেয়ে বড় মানের অবস্থান সংরক্ষণ করে। এই অ্যালগরিদমটিকে বৃহত্তম-মূল্যের অনুক্রমিক অনুসন্ধান বলা হয় এবং নিম্নলিখিত C++ ফাংশন দ্বারা চিত্রিত করা হয়:

```
// "A" আকারের "n" int large(int A[], int n) { int currlarge = 0; // সবচেয়ে বড় উপাদান অবস্থান ধারণ করে

    যদি (A[currlarge] < A[i]) // যদি A[i] বড় হয় //          // প্রতিটি অ্যারের উপাদানের জন্য (int i=1; i<n; i++)
      এর অবস্থান মনে রাখবেন
            currlarge = i; currlarge
    ফেরত;                                                    // বৃহত্তম অবস্থান ফেরত
}
```

এখানে, সমস্যার আকার হল n, A তে সংরক্ষিত পূর্ণসংখ্যার সংখ্যা। মৌলিক কাজ হল একটি পূর্ণসংখ্যার মানকে সবচেয়ে বড় মানের সাথে তুলনা করা।

এ পর্যন্ত দেখা। এটি অনুমান করা যুক্তিসঙ্গত যে দুটি পূর্ণসংখ্যার মান বা অ্যারেতে তাদের অবস্থান নির্বিশেষে এই ধরনের একটি তুলনা করতে একটি নির্দিষ্ট পরিমাণ সময় লাগে।

কারণ চলমান সময়কে প্রভাবিত করে সবচেয়ে গুরুত্বপূর্ণ ফ্যাক্টরটি সাধারণত ইনপুটের আকার, একটি প্রদত্ত ইনপুট আকার n এর জন্য আমরা প্রায়শই T(n) হিসাবে লিখিত n-এর একটি ফাংশন হিসাবে অ্যালগরিদম চালানোর জন্য সময়টি প্রকাশ করি। আমরা সবসময় ধরে নেব T(n) একটি অ-নেতিবাচক মান।

সবচেয়ে বড় ফাংশনে দুটি পূর্ণসংখ্যার তুলনা করার জন্য যে পরিমাণ সময় প্রয়োজন তা এখন c বলা যাক । c এর সুনির্দিষ্ট মান কী হতে পারে তা আমরা এখনই চিন্তা করি না। বা আমরা ভ্যারিয়েবল-এবল i বৃদ্ধি করার জন্য প্রয়োজনীয় সময় নিয়ে উদ্বিগ্ন নই কারণ অ্যারের প্রতিটি মানের জন্য এটি করা আবশ্যক, বা একটি বড় মান পাওয়া গেলে প্রকৃত অ্যাসাইনমেন্টের সময়, বা আরম্ভ করার জন্য নেওয়া সামান্য অতিরিক্ত সময়। currlarge আমরা শুধু অ্যালগরিদম কার্যকর করতে নেওয়া সময়ের জন্য একটি যুক্তিসঙ্গত অ্যাপ-প্রক্সিমেশন চাই। সর্ববৃহৎ চালানোর মোট সময় তাই প্রায় cn, কারণ আমাদের অবশ্যই n তুলনা করতে হবে, প্রতিটি তুলনার খরচ c সময় সহ। আমরা বলি যে ফাংশন বৃহত্তম (এবং সাধারণভাবে বৃহত্তম-মূল্যের অনুক্রমিক অনুসন্ধান অ্যালগরিদম) সমীকরণ দ্বারা প্রকাশিত একটি চলমান সময় রয়েছে

$$T(n) = cn.$$

এই সমীকরণটি সবচেয়ে বড়-মূল্যের অনুক্রমিক অনুসন্ধান অ্যালগরিদমের চলমান সময়ের জন্য বৃদ্ধির হার বর্ণনা করে।

উদাহরণ 3.2 একটি স্টেটমেন্টের চলমান সময় যা একটি ভেরিয়েবলে একটি পূর্ণসংখ্যা অ্যারের প্রথম মান নির্ধারণ করে তা হল প্রথম অ্যারের মানটির মান অনুলিপি করার জন্য প্রয়োজনীয় সময়। আমরা অনুমান করতে পারি যে এই অ্যাসাইনমেন্টটি মান নির্বিশেষে একটি ধ্রুবক সময় নেয়। একটি পূর্ণসংখ্যা অনুলিপি করার জন্য প্রয়োজনীয় সময়কে c1 বলা যাক । একটি সাধারণ কম্পিউটারে অ্যারে যত বড়ই হোক না কেন (মেমরি এবং অ্যারের আকারের জন্য যুক্তিসঙ্গত শর্ত দেওয়া হয়েছে), অ্যারের প্রথম অবস্থান থেকে মানটি কপি করার সময় সর্বদা c1। সুতরাং, এই অ্যালগরিদমের সমীকরণটি সহজ

$$T(n) = c1,$$

নির্দেশ করে যে ইনপুট n এর আকার চলমান সময়ের উপর কোন প্রভাব ফেলে না।
একে বলা হয় স্থির চলমান সময়।

উদাহরণ 3.3 নিম্নলিখিত C++ কোড বিবেচনা করুন:

```
যোগফল = 0;
জন্য (i=1; i<=n; i++)
    (j=1; j<=n; j++) যোগফল++;
```

এই কোড খণ্ডের জন্য চলমান সময় কি? স্পষ্টতই n বড় হলে এটি চালাতে বেশি সময় নেয়। এই উদাহরণের মৌলিক অপারেশন হল পরিবর্তনশীল যোগফলের জন্য বৃদ্ধির ক্রিয়াকলাপ। আমরা অনুমান করতে পারি যে বৃদ্ধির জন্য ধ্রুবক সময় লাগে; এই সময় c2 কল করুন। (আমরা যোগফল শুরু করার জন্য প্রয়োজনীয় সময় উপেক্ষা করতে পারি, এবং লুপ কাউন্টার i এবং j বৃদ্ধি করতে। অনুশীলনে, এই খরচগুলি নিরাপদে সময় c2 এ বান্ডিল করা যেতে পারে।) বৃদ্ধির মোট সংখ্যা

2 হল n . সুতরাং, আমরা বলি যে চলমান সময় হল $T(n) = c_2n^2$.

একটি অ্যালগরিদমের বৃদ্ধির হার হল সেই হার যা অ্যালগরিদমের ইনপুটের আকার বাড়ার সাথে সাথে এর ব্যয় বৃদ্ধি পায়। চিত্র 3.1 ছয়টি সমীকরণের জন্য একটি গ্রাফ দেখায়, প্রতিটি একটি নির্দিষ্ট প্রোগ্রাম বা অ্যালগরিদমের চলমান সময় বর্ণনা করার জন্য। সাধারণ অ্যালগরিদমের প্রতিনিধিত্বকারী বিভিন্ন বৃদ্ধির হার দেখানো হয়েছে। 10n এবং 20n লেবেলযুক্ত দুটি সমীকরণ সরলরেখা দ্বারা গ্রাফ করা হয়েছে। cn-এর বৃদ্ধির হার (c যেকোন ধনাত্মক ধ্রুবকের জন্য) প্রায়ই একটি রৈখিক বৃদ্ধির হার বা চলমান সময় হিসাবে উল্লেখ করা হয়। এর মানে হল যে n-এর মান যত বাড়বে, অ্যালগরিদমের চলমান সময় একই অনুপাতে বৃদ্ধি পাবে। n-এর মান দ্বিগুণ করলে চলমান সময় প্রায় দ্বিগুণ হয়। একটি অ্যালগরিদম যার চলমান-সময়ের সমীকরণে n-এর একটি ফ্যাক্টর 2 সমন্বিত সর্বোচ্চ-ক্রম পদ রয়েছে

একটি দ্বিঘাত বৃদ্ধি হার আছে বলা হয় . চিত্র 3.1-এ, 2n লেবেলযুক্ত লাইনটি একটি সূচকীয় প্রতিনিধিত্ব 2 একটি দ্বিঘাত বৃদ্ধির হার প্রতিনিধিত্ব করে। লাইন 2 বৃদ্ধির হার লেবেল. এই n করে নামটি এসেছে যে সূচকটিতে n উপস্থিত হয়। লাইন লেবেল n! এছাড়াও দ্রুতগতিতে বৃদ্ধি পাচ্ছে।

আপনি চিত্র 3.1 থেকে দেখতে পাচ্ছেন, একটি অ্যালগরিদমের মধ্যে পার্থক্য যার 2 হয় চলমান সময়ের জন্য খরচ হয়েছে T(n) = 10n এবং অন্যটি খরচ T(n) = 2n n বৃদ্ধির সাথে সাথে অসাধারণ। n > 5 এর জন্য, চলমান সময় T(n) = 2n সহ অ্যালগরিদম ইতিমধ্যে 2 অনেক ধীর। 2n-এর তুলনায় 10n-এর একটি বৃহত্তর ধ্রুবক গুণনীয়ক 2 থাকা সত্ত্বেও এটি দেখায় যে একটি সমীকরণের জন্য ধ্রুবক গুণনীয়ক পরিবর্তন করা শুধুমাত্র সেই বিন্দুকে পরিবর্তন করে যেখানে দুটি বক্ররেখা অতিক্রম করে। n > 10 এর জন্য, খরচ T(n) = 2n সহ অ্যালগরিদম T(n) = 20n খরচ সহ অ্যালগরিদমের চেয়ে ধীর । এই গ্রাফটি আরও দেখায় যে T(n) = 5n log n সমীকরণটি T(n) = 10n এবং T(n) = 20n উভয়ের চেয়ে কিছুটা বেশি দ্রুত বৃদ্ধি পায়, কিন্তু T(n) = 2n সমীকরণের মতো এত দ্রুত নয় 2 ধ্রুবক a, b > 1, n logb n বা log n এর চেয়ে দ্রুত

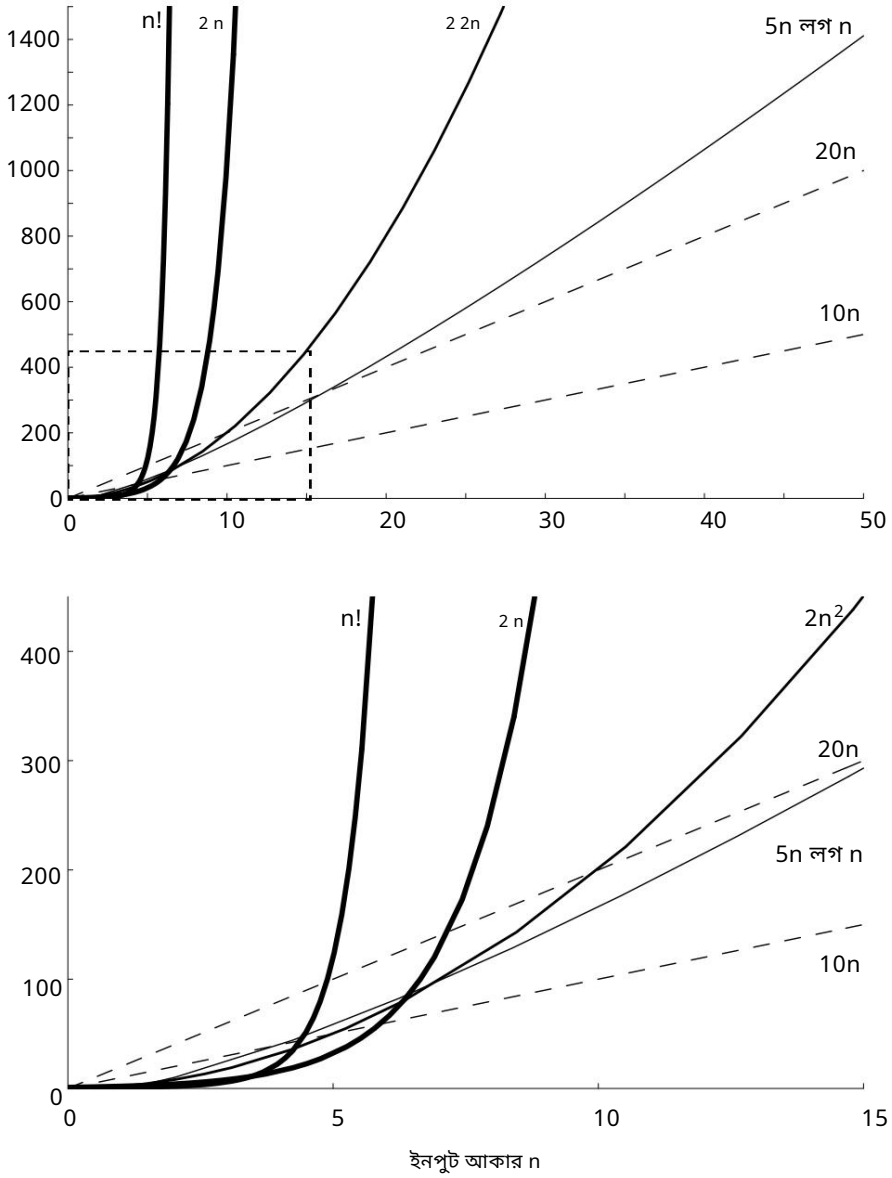
20n এবং 2n চিহ্নিত দুটি বক্ররেখার তুলনা করা

2

2. ক বৃদ্ধি পায়

খঅবশেষে, খরচ T(n) = 2n বা T(n) = n সহ অ্যালগরিদম ! n-এর এমনকি শালীন মানের জন্য নিষেধাজ্ঞামূলকভাবে ব্যয়বহুল। মনে রাখবেন ধ্রুবক a, b 1, a এর জন্য

n n এর চেয়ে দ্রুত বৃদ্ধি পায় খ.



চিত্র 3.1 একটি গ্রাফের দুটি দৃশ্য ছয়টি সমীকরণের জন্য বৃদ্ধির হার চিত্রিত করে।
নীচের দৃশ্যটি উপরের দৃশ্যের নীচের-বাম অংশটি বিস্তারিতভাবে দেখায়। অনুভূমিক-অনুভূমিক অক্ষ ইনপুট আকার উপস্থাপন করে। উল্লম্ব অক্ষ সময়, স্থান, বা প্রতিনিধিত্ব করতে পারে
খরচের অন্য কোনো পরিমাপ।

n	লগ লগ n	লগ n	n	n লগ n	2 n	3 n	2 n
16	2	4	4 2	4 2 · 2 5 = 2	8 2	12 2	2 16
256	3	8	8 2	8 1 1 8 · 2 = 2	16 2	2 24	2 256
1024 □ 3.3		10 10 2		10 10 · 2 13 □ 2	2 20	2 30	2 1024
64K	4	16 16 2		16 16 · 2 20 = 2	2 32	2 48	2 64K
1M □ 4.3 1G □ 4.9		20 20 2		20 20 · 2 24 □ 2	2 40	2 60	2 1 মি
		30 30 2		30 30 · 2 35 □ 2	2 60	2 90	2 1 জি

চিত্র 3.2 বেশিরভাগ কম্পিউটার অ্যালগরিদমের প্রতিনিধি বৃদ্ধির হারের জন্য খরচ।

আমরা বিভিন্ন অ্যালগরিদমের আপেক্ষিক বৃদ্ধির হার সম্পর্কে আরও কিছু অন্তর্দৃষ্টি পেতে পারি
চিত্র 3.2 থেকে। সাধারণ অ্যালগরিদমগুলিতে প্রদর্শিত বেশিরভাগ বৃদ্ধির হার
কিছু প্রতিনিধি ইনপুট আকার সহ দেখানো হয়েছে। আবার, আমরা দেখতে পাই যে
বৃদ্ধির হার একটি অ্যালগরিদম দ্বারা গ্রাস সম্পদের উপর একটি অসাধারণ প্রভাব আছে.

3.2 সেরা, সবচেয়ে খারাপ এবং গড় কেস

n এর ফ্যাক্টরিয়াল বের করার সমস্যাটি বিবেচনা করুন। এই সমস্যার জন্য, শুধুমাত্র আছে
একটি প্রদত্ত "আকার" এর একটি ইনপুট (অর্থাৎ, আকার n এর জন্য শুধুমাত্র একটি একক উদাহরণ রয়েছে
n এর প্রতিটি মান)। এখন আমাদের সবচেয়ে বড়-মূল্যের অনুক্রমিক অনুসন্ধান অ্যালগরিদম বিবেচনা করুন
উদাহরণ 3.1, যা সর্বদা প্রতিটি অ্যারের মান পরীক্ষা করে। এই অ্যালগরিদম কাজ করে
একটি প্রদত্ত আকার n অনেক ইনপুট. যে, যে কোনো প্রদত্ত অনেক সম্ভাব্য অ্যারে আছে
আকার যাইহোক, অ্যালগরিদম যে অ্যারেই দেখুক না কেন, এর খরচ সবসময়ই থাকবে
একই যে এটি সবসময় অ্যারের প্রতিটি উপাদান এক সময় দেখায়.

কিছু অ্যালগরিদমের জন্য, প্রদত্ত আকারের বিভিন্ন ইনপুটগুলির জন্য বিভিন্ন পরিমাণের প্রয়োজন হয়
সময়ের উদাহরণস্বরূপ, n ধারণকারী একটি অ্যারে অনুসন্ধানের সমস্যা বিবেচনা করুন
পূর্ণসংখ্যা একটি নির্দিষ্ট মান K সহ একটি খুঁজে বের করতে (ধরুন যে K হুবহু দেখা যাচ্ছে
একবার অ্যারে)। অনুক্রমিক অনুসন্ধান অ্যালগরিদম প্রথম অবস্থানে শুরু হয়
অ্যারে এবং K পাওয়া না যাওয়া পর্যন্ত প্রতিটি মান ঘুরে দেখে। কে পাওয়া গেলে,
অ্যালগরিদম স্টপ। এটি বৃহত্তম-মূল্যের অনুক্রমিক অনুসন্ধান অ্যালগরিদম থেকে আলাদা
উদাহরণ 3.1, যা সর্বদা প্রতিটি অ্যারের মান পরীক্ষা করে।

অনুক্রমিক অনুসন্ধান অ্যালগ-অরিদমের জন্য সম্ভাব্য চলমান সময়ের বিস্তৃত পরিসর রয়েছে। অ্যারের প্রথম পূর্ণসংখ্যার মান K হতে পারে এবং তাই শুধুমাত্র একটি পূর্ণসংখ্যা
পরীক্ষা করা হয়। এই ক্ষেত্রে চলমান সময় কম। এটি এই জন্য সেরা কেস
অ্যালগরিদম, কারণ অনুক্রমিক অনুসন্ধানের জন্য একের কম দেখা সম্ভব নয়
মান বিকল্পভাবে, যদি অ্যারের শেষ অবস্থানে K থাকে, তাহলে চলমান
সময় অপেক্ষাকৃত দীর্ঘ, কারণ অ্যালগরিদম অবশ্যই n মান পরীক্ষা করবে। এই হল
এই অ্যালগরিদমের জন্য সবচেয়ে খারাপ ঘটনা , কারণ অনুক্রমিক অনুসন্ধান কখনই এর চেয়ে বেশি দেখায় না

n মান। যদি আমরা একটি প্রোগ্রাম হিসাবে অনুক্রমিক অনুসন্ধানটি প্রয়োগ করি এবং n আকারের বিভিন্ন অ্যারেতে এটিকে বহুবার চালাই, বা একই অ্যারের মধ্যে K-এর বিভিন্ন মান অনুসন্ধান করি, আমরা আশা করি যে অ্যালগরিদম মান খুঁজে পাওয়ার আগে অ্যারের মধ্য দিয়ে অর্ধেক যেতে হবে। আমরা খুঁজছি। গড়ে, অ্যালগরিদম প্রায় n/2 মান পরীক্ষা করে। আমরা এই অ্যালগরিদম জন্য গড় ক্ষেত্রে কল .

একটি অ্যালগরিদম বিশ্লেষণ করার সময়, আমাদের কি সেরা, সবচেয়ে খারাপ বা গড় ক্ষেত্রে অধ্যয়ন করা উচিত? সাধারণত আমরা সেরা ক্ষেত্রে আগ্রহী নই, কারণ এটি খুব কমই ঘটতে পারে এবং সাধারণত অ্যালগরিদমের চলমান সময়ের একটি ন্যায্য বৈশিষ্ট্যের জন্য খুব আশাবাদী। অন্য কথায়, সেরা ক্ষেত্রের উপর ভিত্তি করে বিশ্লেষণ অ্যালগরিদমের আচরণের প্রতিনিধি হওয়ার সম্ভাবনা নেই। যাইহোক, এমন বিরল দৃষ্টান্ত রয়েছে যেখানে একটি সেরা-কেস বিশ্লেষণ দরকারী - বিশেষত, যখন সেরা ক্ষেত্রে ঘটার সম্ভাবনা বেশি থাকে। অধ্যায় 7-এ আপনি কিছু উদাহরণ দেখতে পাবেন যেখানে একটি বাছাই অ্যালগরিদমের জন্য সেরা-কেস চলমান সময়ের সুবিধা নেওয়া একটি সেকেন্ডকে আরও দক্ষ করে তোলে।

কিভাবে সবচেয়ে খারাপ ক্ষেত্রে? সবচেয়ে খারাপ ক্ষেত্রে বিশ্লেষণ করার সুবিধা হল যে আপনি নিশ্চিতভাবে জানেন যে অ্যালগরিদমটি অন্তত ভালভাবে সম্পাদন করতে হবে। এটি রিয়েল-টাইম অ্যাপ্লিকেশনের জন্য বিশেষভাবে গুরুত্বপূর্ণ, যেমন কম্পিউটারের জন্য যা একটি এয়ার ট্রাফিক কন্ট্রোল সিস্টেম নিরীক্ষণ করে। এখানে, একটি অ্যালগরিদম ব্যবহার করা গ্রহণযোগ্য হবে না যা বেশিরভাগ সময় n এরোপ্লেনগুলিকে যথেষ্ট দ্রুত পরিচালনা করতে পারে, কিন্তু যখন সমস্ত n এরোপ্লেন একই দিক থেকে আসছে তখন এটি যথেষ্ট দ্রুত কাজ করতে ব্যর্থ হয়।

অন্যান্য অ্যাপ্লিকেশানগুলির জন্য - বিশেষ করে যখন আমরা অনেকগুলি বিভিন্ন ইনপুটে প্রোগ্রাম চালানোর খরচ একত্রিত করতে চাই - সবচেয়ে খারাপ-কেস বিশ্লেষণ-সিস অ্যালগরিদমের কার্যকারিতার একটি প্রতিনিধি পরিমাপ নাও হতে পারে। প্রায়শই আমরা গড়-কেস চলমান সময় জানতে পছন্দ করি। এর মানে হল যে আমরা n আকারের ইনপুটগুলিতে অ্যালগরিদমের সাধারণ আচরণ জানতে চাই। দুর্ভাগ্যবশত, গড়-কেস বিশ্লেষণ সবসময় সম্ভব হয় না। গড়-কেস বিশ্লেষণের জন্য প্রথমে প্রয়োজন যে আমরা বুঝতে পারি যে কীভাবে প্রোগ্রামের প্রকৃত ইনপুটগুলি (এবং তাদের খরচ) প্রোগ্রামে সমস্ত সম্ভাব্য ইনপুটগুলির সেটের ক্ষেত্রে বিতরণ করা হয়। উদাহরণস্বরূপ, এটি আগে বলা হয়েছিল যে অনুক্রমিক অনুসন্ধান অ্যালগরিদম গড়ে অ্যারের মানগুলির অর্ধেক পরীক্ষা করে। এটি শুধুমাত্র তখনই সত্য যদি K মান সহ উপাদানটি অ্যারের যেকোনো অবস্থানে উপস্থিত হওয়ার সমান সম্ভাবনা থাকে। যদি এই অনুমানটি সঠিক না হয়, তাহলে অ্যালগরিদম অগত্যা গড় ক্ষেত্রে অ্যারের মানগুলির অর্ধেক পরীক্ষা করে না।

অনুক্রমিক অনুসন্ধান অ্যালগরিদমে ডেটা বিতরণের প্রভাব সম্পর্কিত আরও আলোচনার জন্য বিভাগ 9.2 দেখুন।

একটি ডেটা বিতরণের বৈশিষ্ট্যগুলি অনেকগুলি অনুসন্ধান অ্যালগরিদমের উপর উল্লেখযোগ্য প্রভাব ফেলে, যেমন হ্যাশিং (বিভাগ 9.4) এবং অনুসন্ধান গাছগুলির উপর ভিত্তি করে (যেমন, বিভাগ 5.4 দেখুন)। তথ্য বিতরণ সম্পর্কে ভুল অনুমান ডিসট্রিবিউশন হতে পারে-

একটি প্রোগ্রামের স্থান বা সময় কর্মক্ষমতা উপর astrous পরিণতি. অস্বাভাবিক ডেটা বিতরণ সুবিধার জন্যও ব্যবহার করা যেতে পারে, যেমনটি ধারা 9.2 এ দেখানো হয়েছে।

সংক্ষেপে, রিয়েল-টাইম অ্যাপ্লিকেশনগুলির জন্য আমরা সম্ভবত একটি অ্যালগরিদমের সবচেয়ে খারাপ-কেস অ্যানাল-ইসিস পছন্দ করব। অন্যথায়, যদি আমরা গড় কেস গণনা করার জন্য আমাদের ইনপুট বিতরণ সম্পর্কে যথেষ্ট জানি তবে আমরা প্রায়শই একটি গড়-কেস বিশ্লেষণ করতে চাই। যদি না হয়, তাহলে আমাদের অবশ্যই সবচেয়ে খারাপ-কেস বিশ্লেষণের অবলম্বন করতে হবে।

3.3 একটি দ্রুত কম্পিউটার, নাকি একটি দ্রুততর অ্যালগরিদম?

কল্পনা করুন যে আপনার সমাধান করার জন্য একটি সমস্যা আছে, এবং আপনি এমন একটি অ্যালগরিদম সম্পর্কে জানেন যার চলমান সময়টি চালানোর জন্য n গুণের সমানুপাতিক। আপনি যদি আপনার বর্তমান কম্পিউটারটিকে একটি নতুন দিয়ে প্রতিস্থাপন করেন যেটি 2 দশগুণ দ্রুত, ^{2}. দুর্ভাগ্যবশত, ফলাফল প্রোগ্রাম দশ লাগে

তাহলে n অ্যালগরিদম কি গ্রহণযোগ্য হবে? যদি সমস্যার আকার একই থাকে, তাহলে সম্ভবত দ্রুততর কম্পিউটার আপনাকে আপনার কাজ দ্রুত সম্পন্ন করার অনুমতি দেবে এমনকি একটি উচ্চ বৃদ্ধির হার থাকা অ্যালগরিদম সহ। কিন্তু একটি মজার জিনিস বেশিরভাগ লোকের সাথে ঘটে যারা একটি দ্রুত কম্পিউটার পান। তারা একই সমস্যা দ্রুত চালায় না। তারা একটি বড় সমস্যা চালান! বলুন যে আপনার পুরানো কম্পিউটারে আপনি 10,000 রেকর্ড বাছাই করতে সন্তুষ্ট ছিলেন কারণ এটি আপনার লাঞ্চ বিরতির সময় কম্পিউটার দ্বারা করা যেতে পারে। আপনার নতুন কম্পিউটারে আপনি একই সময়ে 100,000 রেকর্ড সাজানোর আশা করতে পারেন। আপনি আর শীঘ্রই দুপুরের খাবার থেকে ফিরে আসবেন না, তাই আপনি একটি বড় সমস্যা সমাধান করাই ভালো। এবং যেহেতু নতুন মেশিনটি দশগুণ দ্রুত, আপনি দশগুণ রেকর্ড করতে চান।

যদি আপনার অ্যালগরিদমের বৃদ্ধির হার রৈখিক হয় (অর্থাৎ, যদি ইনপুট আকার n-এ চলমান সময় বর্ণনা করে এমন সমীকরণটি কিছু ধ্রুবক c এর জন্য T(n) = cn হয়), তাহলে নতুন মেশিনে 100,000 রেকর্ড একই সময়ে সাজানো হবে পুরানো মেশিনে 10,000 রেকর্ড। যদি অ্যালগরিদমের বৃদ্ধির হার cn-এর চেয়ে বেশি হয়, যেমন c1n তাহলে আপনি দশগুণ দ্রুত মেশিনে একই পরিমাণে আকারের দশগুণ কোনো সমস্যা করতে পারবেন না। ^{2},

একটি দ্রুত কম্পিউটার দ্বারা একটি নির্দিষ্ট সময়ের মধ্যে কত বড় সমস্যা সমাধান করা যায়? অনুমান করুন যে নতুন মেশিনটি পুরানোটির চেয়ে দশগুণ দ্রুত। বলুন যে পুরানো মেশিনটি এক ঘন্টার মধ্যে n আকারের একটি সমস্যা সমাধান করতে পারে। নতুন মেশিন এক ঘন্টার মধ্যে সমাধান করতে পারে যে সবচেয়ে বড় সমস্যা কি? চিত্র 3.3 দেখায় যে চিত্র 3.1 থেকে পাঁচটি চলমান সময়ের ফাংশনের জন্য দুটি মেশিনে কত বড় সমস্যা সমাধান করা যেতে পারে।

এই টেবিলটি অনেক গুরুত্বপূর্ণ বিষয় তুলে ধরে। প্রথম দুটি সমীকরণ উভয়ই রৈখিক; শুধুমাত্র ধ্রুবক ফ্যাক্টরের মান পরিবর্তিত হয়েছে। উভয় ক্ষেত্রেই, যে যন্ত্রটি দশগুণ দ্রুততর তা দশের গুণনীয়ক দ্বারা সমস্যার আকার বৃদ্ধি করে। অন্য কথায়, যদিও ধ্রুবকের মান সমস্যাটির নিখুঁত আকারকে প্রভাবিত করে যা একটি নির্দিষ্ট সময়ের মধ্যে সমাধান করা যেতে পারে, এটি উন্নতিকে প্রভাবিত করে না

f(n)	n	n	পরিবর্তন করুন n /n 10n
1000 10, 000 n = 10n 10 20n 500 5000 n = 10n 10 1842 □ 10n < n < 10n 7.37 5n লগ n 250 2n21n12 = 312□। n + 3			
2 n	13		---

চিত্র 3.3 সমস্যা আকারের বৃদ্ধি যা একটি নির্দিষ্ট সময়ের মধ্যে একটি কম্পিউটারে চালানো যেতে পারে যা দশগুণ দ্রুত। প্রথম কলামটি চিত্র 3.1-এর পাঁচটি বৃদ্ধির হার সমীকরণের প্রতিটির ডানদিকের দিকগুলিকে তালিকাভুক্ত করে। এই উদাহরণের উদ্দেশ্যে, নির্বিচারে ধরে নিন যে পুরানো মেশিনটি এক ঘন্টায় 10,000 মৌলিক অপারেশন চালাতে পারে। দ্বিতীয় কলামটি n-এর সর্বোচ্চ মান দেখায় যা পুরানো মেশিনে 10,000 মৌলিক অপারেশনে চালানো যেতে পারে। তৃতীয় কলামটি n-এর মান দেখায় সেই সমস্যার জন্য নতুন সর্বোচ্চ মাপের যেটি নতুন মেশিনে একই সময়ে চালানো যেতে পারে যা দশগুণ দ্রুত। 100,000 মৌলিক অপারেশনে চলতে পারে এমন সমস্যার জন্য ভেরিয়েবল n হল সবচেয়ে বড় মাপ। চতুর্থ কলামটি দেখায় কিভাবে n-এর আকার নতুন মেশিনে n-এ পরিবর্তিত হয়েছে। পঞ্চম কলামটি n থেকে n এর অনুপাত হিসাবে সমস্যার আকার বৃদ্ধি দেখায়।

সমস্যা আকার (মূল আকারের অনুপাত হিসাবে) একটি দ্রুত কম্পিউটার দ্বারা অর্জিত। অ্যালগরিদমের বৃদ্ধির হার নির্বিশেষে এই সম্পর্কটি সত্য ধারণ করে: ধ্রুবক কারণগুলি কখনই একটি দ্রুত কম্পিউটার দ্বারা অর্জিত আপেক্ষিক উন্নতিকে প্রভাবিত করে না।

সময় সমীকরণ T(n) = 2n সহ একটি অ্যালগরিদম ^{2} রৈখিক বৃদ্ধির হার সহ একটি অ্যালগরিদম হিসাবে দ্রুত মেশিন থেকে প্রায় দুর্দান্ত উন্নতি পায় না।
দশের গুণনীয়ক দ্বারা উন্নতির পরিবর্তে, উন্নতি হল শুধুমাত্র এর বর্গমূল: □ 10 □ 3.16। এইভাবে, উচ্চ বৃদ্ধির হার সহ অ্যালগরিদম প্রথম স্থানে একটি নির্দিষ্ট সময়ে একটি ছোট সমস্যা সমাধান করে না, এটি একটি দ্রুত কম্পিউটার থেকে কম গতিও পায়। কম্পিউটারগুলি যত দ্রুততর হয়, সমস্যার আকারের বৈষম্য তত বেশি হয়।

বৃদ্ধির হার T(n) = 5n log n সহ অ্যালগরিদম দ্বিঘাত বৃদ্ধির হারের তুলনায় অনেক বেশি পরিমাণে উন্নতি করে, কিন্তু রৈখিক বৃদ্ধির হার সহ অ্যালগো-রিদমগুলির মতো বেশি পরিমাণে নয়।

লক্ষ্য করুন যে অ্যালগরিদমের ক্ষেত্রে বিশেষ কিছু ঘটে যার চলমান সময় দ্রুতগতিতে বৃদ্ধি পায়। চিত্র 3.1-এ, অ্যালগরিদমের বক্ররেখা যার সময় 2 এর সমানুপাতিক তা খুব দ্রুত উঠে যায়। চিত্র 3.3-এ, মেশিনে সমস্যা আকারের দশগুণ দ্রুত ^{n}বৃদ্ধি দেখানো হয়েছে প্রায় n + 3 (সুনির্দিষ্টভাবে বলতে গেলে, এটি n + log2 10)। সূচকীয় বৃদ্ধির হার সহ একটি অ্যালগরিদমের সমস্যা আকারের বৃদ্ধি একটি ধ্রুবক যোগ দ্বারা হয়, গুণিতক ফ্যাক্টর দ্বারা নয়। কারণ n এর পুরানো মান ছিল 13, নতুন সমস্যার আকার হল 16। যদি পরের বছর আপনি আরেকটি কিনবেন

কম্পিউটার এখনও দশগুণ দ্রুত, তারপর নতুন কম্পিউটার (মূল কম্পিউটারের চেয়ে 100 গুণ দ্রুত) শুধুমাত্র 19 আকারের একটি সমস্যা চালাবে। আপনার যদি দ্বিতীয় প্রোগ্রাম থাকে যার বৃদ্ধির হার 2 এবং যার জন্য মূল কম্পিউটারটি একটি সমস্যা চালাতে পারে। এক ঘণ্টায় 1000 সাইজ, একটি মেশিনের চেয়ে দশগুণ দ্রুত গতিতে এক ঘণ্টায় 1003 সাইজের সমস্যা চলতে পারে! এইভাবে, একটি সূচকীয় বৃদ্ধির হার চিত্র 3.3-এ দেখানো অন্যান্য বৃদ্ধির হার থেকে আমূলভাবে ভিন্ন। এই পার্থক্যের তাৎপর্য 17 অধ্যায়ে অন্বেষণ করা হয়েছে।

একটি দ্রুত কম্পিউটার কেনার পরিবর্তে, আপনি যদি একটি নতুন অ্যালগরিদম দিয়ে প্রতিস্থাপন করেন অ্যালগরিদম যার চলমান সময় n চলমান সময়ের সমানুপাতিক n তবে কী হবে তা বিবেচনা করুন লগ n এর সমানুপাতিক। চিত্র 3.1 এর গ্রাফে, একটি নির্দিষ্ট সময় একটি অনুভূমিক রেখা হিসাবে উপস্থিত হবে। যদি আপনার সমস্যা সমাধানের জন্য উপলব্ধ সময়ের রেখাটি প্রশ্নে থাকা দুটি বৃদ্ধির হারের বক্ররেখা যে বিন্দুতে মিলিত হয় তার উপরে হয়, তাহলে অ্যালগরিদম যার চলমান সময় কম দ্রুত বৃদ্ধি পায় তা দ্রুততর। চলমান সময় T(n) = n সহ একটি অ্যালগরিদম 1024 × 1024 = 1, 048, 576 টাইম স্টেপ n = 1024 সাইজের ইনপুটের জন্য প্রয়োজন। চলমান সময় T(n) = n লগ n সহ একটি অ্যালগরিদমের প্রয়োজন 1024 × 10 = n = 1024 আকারের একটি ইনপুটের জন্য 10, 240 টাইম স্টেপ, যা চলমান সময়ের সাথে অ্যালগরিদমের সাথে তুলনা করলে দশের একটি ফ্যাক্টরের চেয়ে অনেক বেশি উন্নতি হয় যখনই n > 58, যদি সাধারণ সমস্যার আকার এই উদাহরণের জন্য 58-এর চেয়ে বড়, তাহলে আপনি দশগুণ দ্রুত কম্পিউটার কেনার পরিবর্তে অ্যালগরিদম পরিবর্তন করা অনেক ভালো হবে। উপরন্তু, আপনি যখন একটি দ্রুততর কম্পিউটার কিনবেন, তখন একটি ধীরগতির বৃদ্ধির হার সহ একটি অ্যালগরিদম নতুন কম্পিউটারে একটি নির্দিষ্ট সময়ে চলতে পারে এমন বড় সমস্যা আকারের ক্ষেত্রে একটি বড় সুবিধা প্রদান করে।

3.4 অ্যাসিম্পোটিক বিশ্লেষণ

চিত্র 3.1-এ 10n লেবেলযুক্ত বক্ররেখার জন্য বৃহত্তর ধ্রুবক থাকা সত্ত্বেও, লেবেলযুক্ত বক্ররেখাটি n = 5 এর তুলনামূলকভাবে ছোট মান দিয়ে 2 2n অতিক্রম করে। যদি আমরা রৈখিক সমীকরণের সামনে ধ্রুবকের মান দ্বিগুণ করি? গ্রাফে দেখানো হিসাবে, 20n লেবেলযুক্ত বক্ররেখাটি 2n একবার n = 10 লেবেলযুক্ত বক্ররেখা অতিক্রম করেছে। রৈখিক বৃদ্ধির হারের জন্য দুটির অতিরিক্ত ফ্যাক্টর খুব বেশি গুরুত্বপূর্ণ নয়; এটি শুধুমাত্র ছেদ বিন্দুর জন্য x-স্থানাঙ্ককে দ্বিগুণ করে। সাধারণভাবে, উভয় সমীকরণের একটি ধ্রুবক ফ্যাক্টরের পরিবর্তন শুধুমাত্র যেখানে দুটি বক্ররেখা অতিক্রম করে সেখানে স্থানান্তরিত হয়, দুটি বক্ররেখা নয়

ক্রস

আপনি যখন একটি দ্রুততর কম্পিউটার বা একটি দ্রুততর কম্পাইলার কিনবেন, তখন প্রদত্ত বৃদ্ধির হারের জন্য একটি নির্দিষ্ট সময়ে চালানো যেতে পারে এমন নতুন সমস্যার আকার একই ফ্যাক্টর দ্বারা বড় হয়, চলমান সময়ের সমীকরণে ধ্রুবক নির্বিশেষে। চলমান সময়ের সমীকরণ ধ্রুবক নির্বিশেষে, বিভিন্ন বৃদ্ধির হার সহ দুটি অ্যালগরিদমের সময় বক্ররেখা এখনও অতিক্রম করে। এই কারণে, আমরা সাধারণত ধ্রুবকগুলিকে উপেক্ষা করি

যখন আমরা চলমান সময় বা অ্যালগরিদমের অন্যান্য সংস্থান প্রয়োজনীয়তার একটি অনুমান চাই। এটি বিশ্লেষণকে সরল করে এবং আমাদের সবচেয়ে গুরুত্বপূর্ণ দিকটি সম্পর্কে চিন্তা করতে রাখে: বৃদ্ধির হার। একে বলা হয় অ্যাসিম্পটোটিক অ্যালগরিদম বিশ্লেষণ।
সুনির্দিষ্টভাবে বলতে গেলে, অ্যাসিম্পোটিক বিশ্লেষণ বলতে একটি অ্যালগরিদমের অধ্যয়নকে বোঝায় কারণ ইনপুট আকার "বড় হয়ে যায়" বা একটি সীমাতে পৌঁছায় (ক্যালকুলাস অর্থে)। যাইহোক, এটি সমস্ত ধ্রুবক কারণকে উপেক্ষা করার জন্য এতটা কার্যকর প্রমাণিত হয়েছে যে বেশিরভাগ অ্যালগরিদম তুলনার জন্য অ্যাসিম্পোটিক বিশ্লেষণ ব্যবহার করা হয়।

ধ্রুবকগুলিকে উপেক্ষা করা সবসময় যুক্তিসঙ্গত নয়। অ্যালগরিদমগুলির তুলনা করার সময় n-এর ছোট মানের উপর চালানো বোঝানো হয়, ধ্রুবকটি একটি বড় প্রভাব ফেলতে পারে। উদাহরণের জন্য, যদি সমস্যাটি হয় ঠিক পাঁচটি রেকর্ডের সংগ্রহ বাছাই করতে, তাহলে হাজার হাজার রেকর্ড বাছাই করার জন্য ডিজাইন করা একটি অ্যালগরিদম সম্ভবত উপযুক্ত নয়, এমনকি যদি এর অ্যাসিম্পোটিক বিশ্লেষণ ভাল কার্যকারিতা নির্দেশ করে। এমন কিছু বিরল ঘটনা আছে যেখানে তুলনার অধীনে দুটি অ্যালগরিদমের ধ্রুবকগুলি 1000 বা তার বেশি একটি ফ্যাক্টর দ্বারা আলাদা হতে পারে, যার ফলে একটি বড় ধ্রুবকের কারণে বেশিরভাগ উদ্দেশ্যে কম বৃদ্ধির হার অকার্যকর হয়ে ওঠে। অ্যাসিম্পোটিক বিশ্লেষণ হল অ্যালগরিদম রিসোর্স খরচের জন্য "খামের পিছনের" অনুমানের একটি রূপ। এটি চলমান সময় বা একটি অ্যালগরিদমের অন্যান্য সংস্থান চাহিদাগুলির একটি সরলীকৃত মডেল সরবরাহ করে। এই সরলীকরণটি সাধারণত আপনাকে আপনার অ্যালগরিদমের আচরণ বুঝতে সাহায্য করে। বিরল পরিস্থিতিতে যেখানে ধ্রুবক গুরুত্বপূর্ণ তা অ্যাসিম্পোটিক বিশ্লেষণের সীমাবদ্ধতা সম্পর্কে সচেতন থাকুন।

3.4.1 উচ্চ সীমা

একটি অ্যালগরিদমের চলমান-সময় সমীকরণ বর্ণনা করতে বেশ কয়েকটি পদ ব্যবহার করা হয়।
এই পদগুলি — এবং তাদের সংশ্লিষ্ট চিহ্নগুলি — অ্যালগরিদমের আচরণের কোন দিকটি বর্ণনা করা হচ্ছে তা সঠিকভাবে নির্দেশ করে। একটি হল অ্যালগরিদমের চলমান সময়ের বৃদ্ধির জন্য উপরের সীমা । এটি উচ্চ বা সর্বোচ্চ বৃদ্ধির হার নির্দেশ করে যা অ্যালগরিদম থাকতে পারে।

একটি অ্যালগরিদমের উপরের সীমা সম্পর্কে কোনো বিবৃতি তৈরি করতে, আমরা অবশ্যই n আকারের কিছু শ্রেণির ইনপুট সম্পর্কে এটি তৈরি করব। আমরা প্রায় সবসময় সেরা-কেস, গড়-কেস, বা সবচেয়ে খারাপ-কেস ইনপুটগুলিতে এই উপরের সীমা পরিমাপ করি। সুতরাং, আমরা পারি না।" আমরা বলতে হবে
বলুন, "এই অ্যালগরিদমটির n এর বৃদ্ধির হারের সাথে একটি উচ্চ সীমা রয়েছে, যেমন 2
"এই অ্যালগরিদমটি n গড় ক্ষেত্রে এর বৃদ্ধির হারের একটি উচ্চ সীমাবদ্ধ।" 2

যেহেতু "f(n) এর বৃদ্ধির হারের সাথে একটি উচ্চ সীমাবদ্ধ" বাক্যাংশটি দীর্ঘ এবং প্রায়শই অ্যালগরিদম নিয়ে আলোচনা করার সময় ব্যবহৃত হয়, আমরা একটি বিশেষ স্বরলিপি গ্রহণ করি, যাকে বলা হয় বিগ-ওহ স্বরলিপি। যদি একটি অ্যালগরিদমের বৃদ্ধির হারের (যেমন, সবচেয়ে খারাপ ক্ষেত্রে) উপরের সীমা f(n) হয়, তাহলে আমরা লিখব যে এই অ্যালগরিদমটি "সেট O(f(n)) সবচেয়ে খারাপ ক্ষেত্রে" (বা শুধু "O(f(n)) সবচেয়ে খারাপ ক্ষেত্রে")। উদাহরণস্বরূপ, যদি n 2 হিসাবে বৃদ্ধি পায়

সবচেয়ে খারাপ ইনপুটের জন্য T(n) (আমাদের অ্যালগরিদমের চলমান সময়) হিসাবে দ্রুত , আমরা বলব অ্যালগরিদম হল "O(n) তে ^{2}) সবচেয়ে খারাপ অবস্থায়।"

নিচের একটি ঊর্ধ্ব সীমার জন্য একটি সুনির্দিষ্ট সংজ্ঞা। T(n) অ্যালগরিদমের সত্যিকারের চলমান সময়কে উপস্থাপন করে। f(n) হল উপরের বাউন্ডের জন্য কিছু এক্সপ্রেশন।

T(n) একটি অ-ঋণাত্মক মূল্যবান ফাংশনের জন্য , T(n) সেট O(f(n)) এ আছে যদি দুটি ধনাত্মক ধ্রুবক c এবং n0 থাকে যাতে T(n) □ cf(n) সব n > এর জন্য n0

ধ্রুবক n0 হল n-এর ক্ষুদ্রতম মান যার জন্য একটি উপরের সীমার দাবিটি সত্য। সাধারণত n0 ছোট হয়, যেমন 1, কিন্তু হওয়ার দরকার নেই। আপনি অবশ্যই কিছু ধ্রুবক সি বাছাই করতে সক্ষম হবেন, তবে সি এর মান আসলে কী তা এটি অপ্রাসঙ্গিক।
অন্য কথায়, সংজ্ঞাটি বলে যে প্রশ্নযুক্ত ধরণের সমস্ত ইনপুটগুলির জন্য (যেমন n আকারের সমস্ত ইনপুটের জন্য সবচেয়ে খারাপ ক্ষেত্রে) যেগুলি যথেষ্ট বড় (যেমন, n > n0), অ্যালগরিদম সর্বদা cf(এর চেয়ে কম সময়ে কার্যকর করে) n) কিছু ধ্রুবকের জন্য পদক্ষেপ গ.

উদাহরণ 3.4 পূর্ণসংখ্যার অ্যারেতে একটি নির্দিষ্ট-নির্দিষ্ট মান খোঁজার জন্য অনুক্রমিক অনুসন্ধান অ্যালগরিদম বিবেচনা করুন। যদি অ্যারেতে একটি মান পরিদর্শন এবং পরীক্ষা করার জন্য cs ধাপের প্রয়োজন হয় যেখানে cs একটি ধনাত্মক সংখ্যা, এবং যদি আমরা যে মানটি অনুসন্ধান করি সেটি ar-ray-এর যেকোনো অবস্থানে উপস্থিত হওয়ার সমান সম্ভাবনা থাকে, তাহলে গড় ক্ষেত্রে T(n) = csn/ 2. n > 1, csn/2 □ csn- এর সমস্ত মানের জন্য। সুতরাং, সংজ্ঞা অনুসারে, T(n) n0 = 1 এবং এর জন্য O(n) এ রয়েছে
c = cs.

উদাহরণ 3.5 একটি নির্দিষ্ট অ্যালগরিদমের জন্য, T(n) = c1n এরেজ কেস ^{2} এভিতে + c2n- যেখানে c1 এবং c2 ধনাত্মক সংখ্যা। তারপর, সব n > 1 এর জন্য c1n । তাই, c = ^{2} + c2n □ $_{2\ c1n}$ + c2n ^{2} □ (c1 + c2)n ^{2} c1 + c2 এর জন্য T(n) □ cn2 ,
এবং n0 = 1. অতএব, T(n) O(n) এ রয়েছে ^{2}) সংজ্ঞা দ্বারা।

উদাহরণ 3.6 একটি অ্যারের প্রথম অবস্থান থেকে একটি ভেরিয়েবলে মান বরাদ্দ করা অ্যারের আকার নির্বিশেষে ধ্রুবক সময় নেয়। এইভাবে, T(n) = c (সর্বোত্তম, সবচেয়ে খারাপ এবং গড় ক্ষেত্রে)। আমরা এই ক্ষেত্রে বলতে পারি যে T(n) O(c) এ রয়েছে। যাইহোক, এটা বলা প্রথাগত যে একটি অ্যালগরিদম যার চলমান সময়ের একটি ধ্রুবক উপরের সীমা রয়েছে O(1) তে।

শুধুমাত্র O(f(n)) তে কিছু আছে তা জেনেই কেবল কতটা খারাপ জিনিস পেতে পারে তা বলে। সম্ভবত জিনিসগুলি এতটা খারাপ নয়। কারণ আমরা জানি অনুক্রমিক অনুসন্ধান)। কিন্তু O(n) সবচেয়ে খারাপ ক্ষেত্রে, এটা বলাও সত্য যে অনুক্রমিক অনুসন্ধান O(n) এ ^{2}

অনুক্রমিক অনুসন্ধান বড় n এর জন্য ব্যবহারিক, এমনভাবে যা অন্য কারো জন্য সত্য নয়)। আমরা সর্বদা টাইট (সর্বনিম্ন) সম্ভাব্য n^2 একটি অ্যালগরিদমের চলমান সময় সংজ্ঞায়িত করতে চাই উপরের বাউন্ড সহ O(n-এ অ্যালগরিদম। সুতরাং, আমরা বলতে পছন্দ করি যে সিকোয়েন্স-টিয়াল অনুসন্ধান O(n) এ রয়েছে। এটি ব্যাখ্যা করে যে কেন বাক্যাংশটি "O(f(n)) তে রয়েছে" অথবা “O(f(n))” বা “= O(f(n))-এর পরিবর্তে “ O(f(n))” ব্যবহার করা হয়। ওহ স্বরলিপি O(n) এ O(n)।

n^2), কিন্তু O(n^2) এটি না

3.4.2 নিম্ন সীমানা

বিগ-ওহ স্বরলিপি একটি উপরের সীমাকে বর্ণনা করে। অন্য কথায়, বিগ-ওহ স্বরলিপি কিছু সম্পদের (সাধারণত সময়) সর্বাধিক পরিমাণের দাবি করে যা একটি অ্যালগরিদম দ্বারা n আকারের কিছু শ্রেণীর ইনপুটগুলির জন্য প্রয়োজন হয় (সাধারণত সবচেয়ে খারাপ ইনপুট, সমস্ত সম্ভাব্য ইনপুটগুলির গড়, বা এই ধরনের সেরা ইনপুট)।

অনুরূপ স্বরলিপি একটি সম্পদের ন্যূনতম পরিমাণ বর্ণনা করতে ব্যবহৃত হয় যা একটি অ্যালগ-অরিদমকে কিছু শ্রেণীর ইনপুটের জন্য প্রয়োজন। বড়-ওহ স্বরলিপির মতো, এটি অ্যালগরিদমের বৃদ্ধির হারের একটি পরিমাপ। বিগ-ওহ স্বরলিপির মতো, এটি যে কোনও সংস্থানের জন্য কাজ করে, তবে আমরা প্রায়শই প্রয়োজনীয় সময়ের সর্বনিম্ন পরিমাণ পরিমাপ করি। এবং আবার, বড়-ওহ নো-টেশনের মতো, আমরা কিছু নির্দিষ্ট শ্রেণীর ইনপুটগুলির জন্য প্রয়োজনীয় সংস্থান পরিমাপ করছি: আকার n-এর সবচেয়ে খারাপ-, গড়-, বা সেরা-কেস ইনপুট।

একটি অ্যালগরিদমের নিম্ন সীমা (বা একটি সমস্যা, যেমনটি পরে ব্যাখ্যা করা হয়েছে) Ω চিহ্ন দ্বারা চিহ্নিত করা হয়, "বিগ-ওমেগা" বা শুধু "ওমেগা"। Ω-এর নিম্নলিখিত সংজ্ঞাটি বড়-ওহ-এর সংজ্ঞার সাথে প্রতিসম।

T(n) একটি অ-ঋণাত্মক মূল্যবান ফাংশনের জন্য , T(n) Ω(g(n)) সেটে থাকে যদি দুটি ধনাত্মক ধ্রুবক c এবং n0 থাকে যাতে সমস্ত n-এর জন্য T(n) cg(n) 1 থাকে > n0।

1এর জন্য একটি বিকল্প (অ-সমতুল্য) সংজ্ঞা হল

T(n) Ω(g(n)) সেটে থাকে যদি একটি ধনাত্মক ধ্রুবক c থাকে যেমন n-এর জন্য অসীম সংখ্যক মানের জন্য T(n) cg(n)।

এই সংজ্ঞাটি বলে যে একটি "আকর্ষণীয়" সংখ্যক ক্ষেত্রে, অ্যালগরিদম কমপক্ষে cg(n) সময় নেয়। উল্লেখ্য যে এই সংজ্ঞাটি বড়-ওহ এর সংজ্ঞার সাথে প্রতিসম নয়। g(n) একটি নিম্ন সীমাবদ্ধ হওয়ার জন্য, এই সংজ্ঞার প্রয়োজন হয় না যে T(n) cg(n) কিছু ধ্রুবকের চেয়ে বড় n এর সমস্ত মানের জন্য। এটি শুধুমাত্র প্রয়োজন যে এটি প্রায়ই যথেষ্ট ঘটবে, বিশেষ করে এটি n-এর জন্য অসীম সংখ্যক মানের জন্য ঘটবে। এই বিকল্প সংজ্ঞার জন্য অনুপ্রেরণা নিম্নলিখিত উদাহরণে পাওয়া যাবে।

অনুমান করুন একটি নির্দিষ্ট অ্যালগরিদমের নিম্নলিখিত আচরণ রয়েছে:

$$T(n) = \begin{cases} \text{সব জোড় } n \\ n^2 \, 1/100 \text{ সব জোড় } n \text{ 0 এর জন্য} \end{cases}$$

থেকে, c = 1/100 এর জন্য n (অর্থাৎ, সমস্ত জোড় n) $n^2/100 \quad 1/100 \quad n^2$ সকলের জন্য জোড় n 0। সুতরাং, একটি অসীম সংখ্যার জন্য T(n) cn2 এই সংজ্ঞা এর মানের n 100n । অতএব, T(n) Ω(n^2) সংজ্ঞা দ্বারা।

উদাহরণ 3.7 অনুমান করুন T(n) = c1n ^{2} c1 এবং c2 > 0 এর জন্য + c2n । তারপর,

$$2\ c1n \quad 2 + c2n \square c1n$$

সকলের জন্য n > 1। সুতরাং, c = c1 এবং n0 = 1 এর জন্য T(n) □ cn2। অতএব, T(n) Ω(n) এ রয়েছে ^{2}) সংজ্ঞা দ্বারা।

এটাও সত্য যে উদাহরণ 3.7 এর সমীকরণটি Ω(n) এ রয়েছে। যাইহোক, বিগ-ওহ স্বরলিপির মতো, আমরা "আঁটসাঁট" পেতে চাই (Ω স্বরলিপির জন্য, বৃহত্তম) সম্ভাব্য আবদ্ধ। এইভাবে, আমরা বলতে পছন্দ করি যে এই চলমান সময়টি Ω(n এ পূর্ণসংখ্যার একটি অ্যারের মধ্যে K মান খুঁজে পেতে অনুক্রমিক^2)

অনুসন্ধান অ্যালগরিদমটি স্মরণ করুন। গড় এবং সবচেয়ে খারাপ ক্ষেত্রে এই অ্যালগরিদমটি Ω(n), কারণ উভয় ক্ষেত্রেই গড় এবং সবচেয়ে খারাপ ক্ষেত্রে আমাদের অবশ্যই কমপক্ষে cn মান পরীক্ষা করতে হবে (যেখানে c গড় ক্ষেত্রে 1/2 এবং সবচেয়ে খারাপ ক্ষেত্রে 1)।

3.4.3 Θ স্বরলিপি

বড়-ওহ এবং Ω-এর সংজ্ঞাগুলি আমাদেরকে একটি অ্যালগরিদমের জন্য উপরের সীমা বর্ণনা করার উপায় দেয় (যদি আমরা n আকারের একটি নির্দিষ্ট শ্রেণীর ইনপুটের সর্বাধিক ব্যয়ের জন্য একটি সমীকরণ খুঁজে পাই) এবং একটি অ্যালগরিদমের জন্য নিম্ন সীমা (যদি আমরা n আকারের একটি নির্দিষ্ট শ্রেণীর ইনপুটগুলির জন্য সর্বনিম্ন খরচের জন্য একটি সমীকরণ খুঁজে পেতে পারে)। যখন একটি ধ্রুবক গুণকের মধ্যে উপরের এবং নীচের সীমা একই থাকে, তখন আমরা Θ (বিগ-থিটা) স্বরলিপি ব্যবহার করে এটি নির্দেশ করি। একটি অ্যালগরিদমকে Θ(h(n)) বলা হয় যদি এটি O(h(n)) এ থাকে এবং এটি Ω(h(n)) থাকে। মনে রাখবেন যে আমরা Θ স্বরলিপির জন্য "in" শব্দটি বাদ দিই, কারণ একই Θ সহ দুটি সমীকরণের জন্য একটি কঠোর সমতা রয়েছে। অন্য কথায়, যদি f(n) হয় Θ(g(n)), তাহলে g(n) হল Θ(f(n))।

কারণ অনুক্রমিক অনুসন্ধান অ্যালগরিদম উভয়ই O(n) এবং Ω(n) তে
গড় ক্ষেত্রে, আমরা বলি গড় ক্ষেত্রে এটি Θ(n)।

একটি অ্যালগরিদমের জন্য সময়ের প্রয়োজনীয়তা বর্ণনা করে একটি বীজগণিত সমীকরণ দেওয়া হলে, উপরের এবং নিম্ন সীমাগুলি সর্বদা মিলিত হয়। কারণ কিছু অর্থে আমাদের কাছে অ্যালগরিদমের জন্য একটি নিখুঁত বিশ্লেষণ রয়েছে, যা চলমান-সময় সমীকরণ দ্বারা মূর্ত। জন্য

T(n) এর এই সমীকরণের জন্য , এটা সত্য যে n আকারের সমস্ত ইনপুট কমপক্ষে cn সময় নেয়। কিন্তু একটি অসীম)।
n আকারের ইনপুটগুলির সংখ্যা cn2 সময় নেয়, তাই আমরা বলতে চাই যে অ্যালগরিদমটি Ω(n) এ রয়েছে দুর্ভাগ্যবশত, আমাদের ^{2}
প্রথম সংজ্ঞাটি ব্যবহার করলে Ω(n) এর একটি নিম্ন সীমা পাওয়া যাবে কারণ c এবং ধ্রুবক বাছাই করা সম্ভব নয় n0 যেমন T(n) □ cn2 সব n > n0 এর ফলে
Ω(n-এর নিম্ন সীমানায় স্পষ্টতই, ^{2}) এই অ্যালগরিদমের জন্য, যা সাধারণ জ্ঞানের সাথে আরও ঘনিষ্ঠভাবে ফিট বলে মনে হয়। ফরটু-কয়েকটি বাস্তব অ্যালগরিদম বা কম্পিউটার প্রোগ্রাম এই উদাহরণের রোগগত আচরণ প্রদর্শন করে।
Ω এর জন্য আমাদের প্রথম সংজ্ঞা সাধারণত প্রত্যাশিত ফলাফল দেয়।

আপনি এই আলোচনা থেকে দেখতে পাচ্ছেন, অ্যাসিম্পোটিক বাউন্ড নোটেশন প্রকৃতির নিয়ম নয়। এটি শুধুমাত্র একটি শক্তিশালী মডেলিং টুল যা অ্যালগরিদমের আচরণ বর্ণনা করতে ব্যবহৃত হয়।

অনেক অ্যালগরিদম (অথবা প্রোগ্রাম হিসাবে তাদের ইনস্ট্যান্টেশন), এটি তাদের রানটাইম আচরণ সংজ্ঞায়িত করে এমন সমীকরণ নিয়ে আসা সহজ। এই বইটিতে উপস্থাপিত বেশিরভাগ অ্যালগরিদমগুলি ভালভাবে বোঝা যায় এবং আমরা প্রায় সবসময়ই তাদের জন্য একটি Θ বিশ্লেষণ দিতে পারি।

যাইহোক, অধ্যায় 17 অ্যালগরিদমগুলির একটি সম্পূর্ণ শ্রেণী নিয়ে আলোচনা করে যার জন্য আমাদের কোন Θ বিশ্লেষণ নেই, শুধুমাত্র কিছু অসন্তুষ্ট বড়-ওহ এবং Ω বিশ্লেষণ। ব্যায়াম 3.14 একটি সংক্ষিপ্ত, সহজ প্রোগ্রাম খণ্ড উপস্থাপন করে যার জন্য বর্তমানে কেউ প্রকৃত উপরের বা নীচের সীমানা জানে না।

যদিও কিছু পাঠ্যপুস্তক এবং প্রোগ্রামাররা আকস্মিকভাবে বলবেন যে একটি অ্যালগরিদম কিছু খরচ ফাংশনের "অর্ডার অফ" বা "বিগ-ওহ", এটি সাধারণত বড়-ওহ স্বরলিপির পরিবর্তে Θ স্বরলিপি ব্যবহার করা ভাল যখনই আমাদের কাছে পর্যাপ্ত জ্ঞান থাকে। অরিদম নিশ্চিত হতে হবে যে উপরের এবং নীচের সীমাগুলি প্রকৃতপক্ষে মেলে। এই বই জুড়ে, Θ স্বরলিপি বড়-ওহ স্বরলিপিকে অগ্রাধিকার হিসাবে ব্যবহার করা হবে যখনই আমাদের জ্ঞানের অবস্থা এটি সম্ভব করে। নির্দিষ্ট অ্যালগরিদম বিশ্লেষণ করার আমাদের ক্ষমতার সীমাবদ্ধতার জন্য বড়-ওহ বা Ω স্বরলিপি ব্যবহারের প্রয়োজন হতে পারে। বিরল ক্ষেত্রে যখন আলোচনাটি একটি সমস্যা বা অ্যালগরিদমের উপরের বা নীচের সীমা সম্পর্কে স্পষ্টভাবে হয়, তখন সংশ্লিষ্ট স্বরলিপিটি Θ স্বরলিপির অগ্রাধিকারে ব্যবহার করা হবে।

3.4.4 সরলীকরণ নিয়ম

একবার আপনি একটি অ্যালগরিদমের জন্য চলমান-সময়ের সমীকরণ নির্ধারণ করলে, সমীকরণ থেকে বড়-ওহ, Ω এবং Θ অভিব্যক্তিগুলি বের করা সত্যিই একটি সহজ বিষয়। আপনাকে অ্যাসিম্পোটিক বিশ্লেষণের আনুষ্ঠানিক সংজ্ঞা অবলম্বন করার দরকার নেই। পরিবর্তে, আপনি সহজ ফর্ম নির্ধারণ করতে নিম্নলিখিত নিয়ম ব্যবহার করতে পারেন।

1. যদি f(n) O(g(n)) এ থাকে এবং g(n) O(h(n) তে থাকে, তাহলে f(n) O(h(n) এ থাকে।
2. যদি f(n) কোন ধ্রুবক k > 0 এর জন্য O(kg(n)) থাকে, তাহলে f(n) O(g(n)) এ থাকে।
3. যদি f1(n) O(g1(n)) এ থাকে এবং f2(n) O(g2(n) এ থাকে, তাহলে f1(n) + f2(n) O(max(g1(n) তে থাকে।), g2(n)))।
4. যদি f1(n) O(g1(n)) এ থাকে এবং f2(n) O(g2(n) তে থাকে, তাহলে f1(n)f2(n) O(g1(n)g2(এ থাকে) n))।

প্রথম নিয়মটি বলে যে যদি কিছু ফাংশন g(n) আপনার খরচ ফাংশনের জন্য একটি ঊর্ধ্ব বাউন্ড হয়, তাহলে g(n) এর জন্য যেকোন ঊর্ধ্ব বাউন্ডটিও আপনার খরচ ফাংশনের জন্য একটি উপরের সীমা। একটি অনুরূপ সম্পত্তি Ω স্বরলিপির জন্য সত্য: যদি g(n) আপনার খরচ ফাংশনের জন্য একটি নিম্ন সীমা হয়, তাহলে g(n) এর জন্য যেকোনো নিম্ন সীমাও আপনার খরচ ফাংশনের জন্য একটি নিম্ন সীমা। একইভাবে Θ স্বরলিপির জন্য।

নিয়ম (2) এর তাৎপর্য হল যে আপনি বড়-ওহ স্বরলিপি ব্যবহার করার সময় আপনার সমীকরণের যেকোনো গুণক ধ্রুবককে উপেক্ষা করতে পারেন। এই নিয়মটি Ω এবং Θ স্বরলিপির জন্যও সত্য।

নিয়ম (3) বলে যে একটি প্রোগ্রামের দুটি অংশ ক্রমানুসারে চালানো হয় (হোক দুটি বিবৃতি বা কোডের দুটি বিভাগ), আপনাকে কেবলমাত্র আরও ব্যয়বহুল অংশ বিবেচনা করতে হবে। এই নিয়মটি Ω এবং Θ স্বরলিপির ক্ষেত্রেও প্রযোজ্য: উভয়ের জন্য, আপনাকে শুধুমাত্র আরও ব্যয়বহুল অংশ বিবেচনা করতে হবে।

নিয়ম (4) প্রোগ্রামে সাধারণ লুপ বিশ্লেষণ করতে ব্যবহৃত হয়। যদি কিছু ক্রিয়া কিছু সংখ্যক বার পুনরাবৃত্তি করা হয় এবং প্রতিটি পুনরাবৃত্তির একই খরচ হয়, তাহলে মোট খরচ হল ক্রিয়াটি যতবার সংঘটিত হয় তার সংখ্যা দ্বারা গুণ করা হয়।
এই নিয়মটি Ω এবং Θ স্বরলিপির ক্ষেত্রেও প্রযোজ্য।

প্রথম তিনটি নিয়ম সম্মিলিতভাবে গ্রহণ করে, আপনি যেকোন খরচ ফাংশনের জন্য অ্যাসিম্পোটিক বৃদ্ধির হার নির্ধারণ করতে সমস্ত ধ্রুবক এবং সমস্ত নিম্ন-ক্রম পদ উপেক্ষা করতে পারেন।
ধ্রুবক উপেক্ষা করার সুবিধা এবং বিপদগুলি এই বিভাগের শুরুতে আলোচনা করা হয়েছিল। একটি অ্যাসিম্পোটিক বিশ্লেষণ সম্পাদন করার সময় নিম্ন-ক্রমের শর্তাবলী উপেক্ষা করা যুক্তিসঙ্গত। উচ্চ-অর্ডার পদগুলি শীঘ্রই নিম্ন-অর্ডার পদগুলিকে তাদের মোট খরচে অবদানের জন্য অদলবদল করে, কারণ n বড় হয়। এইভাবে, যদি T(n) = 3n হয় তাহলে $4\,2 + 5n$,
T(n) O(n) এ থাকে ^{4}) n শব্দটি ^{2} মোট খরচে তুলনামূলকভাবে সামান্য অবদান রাখে।

এই বইয়ের বাকি অংশ জুড়ে, একটি প্রোগ্রাম বা অ্যালগরিদমের খরচ আলোচনা করার সময় এই সরলীকরণ নিয়মগুলি ব্যবহার করা হয়।

3.4.5 শ্রেণীবিন্যাস ফাংশন

প্রদত্ত ফাংশন f(n) এবং g(n) যার বৃদ্ধির হার বীজগণিতীয় সমীকরণ হিসাবে প্রকাশ করা হয়, আমরা নির্ধারণ করতে চাই যে একটি অন্যটির চেয়ে দ্রুত বৃদ্ধি পায় কিনা। এটি করার সর্বোত্তম উপায় হল দুটি ফাংশনের সীমা গ্রহণ করা যখন n অসীমের দিকে বৃদ্ধি পায়,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}.$$

যদি সীমা এ যায়, তাহলে f(n) Ω(g(n)) এ থাকে কারণ f(n) দ্রুত বৃদ্ধি পায়। যদি সীমা শূন্যে যায়, তাহলে f(n) O(g(n)) এ থাকে কারণ g(n) দ্রুত বৃদ্ধি পায়। যদি সীমাটি শূন্য ব্যতীত অন্য কোন ধ্রুবকের দিকে যায়, তাহলে f(n) = Θ(g(n)) কারণ উভয়ই বৃদ্ধি পায়
একই হার।

উদাহরণ 3.8 যদি f(n) = 2n লগ n এবং g(n) = n Ω(g(n)), অথবা ^{2}, O(g(n) তে f(n) আছে, Θ(g(n))? কারণ

$$\frac{2\,n}{2n \text{ লগ } n} = \frac{n}{2 \text{ লগ } n},$$

আমরা তা সহজেই দেখতে পাই

$$\lim_{n\to\infty} \frac{2\,n}{2n \text{ লগ } n} = \infty$$

কারণ n 2 লগ n এর চেয়ে দ্রুত বৃদ্ধি পায়। এইভাবে, এন ^{2} Ω(2n log n) এ আছে।

3.5 একটি প্রোগ্রামের জন্য চলমান সময় গণনা করা

এই বিভাগটি বেশ কয়েকটি সাধারণ কোড খণ্ডের বিশ্লেষণ উপস্থাপন করে।

উদাহরণ 3.9 আমরা একটি পূর্ণসংখ্যা ভেরিয়েবলের একটি সাধারণ অ্যাসাইনমেন্ট স্টেটমেন্টের বিশ্লেষণ দিয়ে শুরু করি।

```
a = b;
```

কারণ অ্যাসাইনমেন্ট বিবৃতিতে ধ্রুবক সময় লাগে, এটি Θ(1)।

উদাহরণ 3.10 লুপের জন্য একটি সহজ বিবেচনা করুন । যোগফল = 0; (i=1; i<=n;

i++) যোগফল +=
n;

প্রথম লাইন হল Θ(1)। জন্য লুপ n বার পুনরাবৃত্তি হয়. তৃতীয় লাইনটি ধ্রুবক সময় নেয় তাই, ধারা 3.4.4 এর নিয়ম (4) সরলীকরণ করে, লুপ তৈরির দুটি লাইন কার্যকর করার জন্য মোট খরচ হল Θ(n)। নিয়ম অনুসারে (3), পুরো কোড খণ্ডের খরচও Θ(n)।

উদাহরণ 3.11 আমরা এখন লুপগুলির জন্য বেশ কয়েকটি সহ একটি কোড খণ্ড বিশ্লেষণ করি , যার মধ্যে কিছু নেস্ট করা আছে।

```
যোগফল = 0; // ফার্স্ট ফর লুপ ফর (i=1;
i<=n; i++) for
                (j=1; j<=i; j++) // একটি ডাবল লুপ

            যোগফল+
+; জন্য (k=0; k<n; k++)                    // লুপের জন্য সেকেন্ড
    A[k] = k;
```

এই কোড খণ্ডটির তিনটি পৃথক বিবৃতি রয়েছে: প্রথম অ্যাসাইনমেন্ট বিবৃতি এবং দুটি লুপের জন্য । আবার অ্যাসাইনমেন্ট স্টেটমেন্টে ধ্রুবক সময় লাগে; এটা c1 কল . লুপের জন্য দ্বিতীয়টি Exam-ple 3.10-এর মত এবং c2n = Θ(n) সময় নেয়।

লুপের জন্য প্রথমটি একটি ডবল লুপ এবং একটি বিশেষ কৌশল প্রয়োজন। আমরা লুপের ভেতর থেকে বাইরের দিকে কাজ করি। রাশির যোগফল++ ধ্রুবক সময়ের প্রয়োজন; এটা c3 কল . কারণ লুপের ভিতরেরটি i বার চালানো হয়, নিয়ম (4) সরলীকরণ করে এটির খরচ c3i হয়েছে। লুপের জন্য বাইরেরটি n বার কার্যকর করা হয়,

কিন্তু প্রতিবার ভিতরের লুপের খরচ আলাদা কারণ প্রতিবার i পরিবর্তন করার সাথে সাথে এটির c3i খরচ হয়। আপনার দেখা উচিত যে বাইরের লুপের প্রথম সঞ্চালনের জন্য, i হল 1। বাইরের লুপের দ্বিতীয় সঞ্চালনের জন্য, i হল 2। প্রতিবার বাইরের লুপের মাধ্যমে, আমি একটি বড় হয়ে উঠি, লুপের মাধ্যমে শেষ সময় পর্যন্ত যখন i = n. এইভাবে, লুপের মোট খরচ 1 থেকে n পর্যন্ত পূর্ণসংখ্যার যোগফলের c3 গুণ। সমীকরণ 2.1 থেকে, আমরা তা জানি

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2},$$

যা Θ(n)। ^{2}) নিয়ম সরলীকরণ করে (3), Θ(c1 + c2n + c3n ^{2}) সহজভাবে Θ(n ^{2}

উদাহরণ 3.12 নিম্নলিখিত দুটি কোড খণ্ডের জন্য অ্যাসিম্পোটিক বিশ্লেষণ তুলনা করুন:

```
যোগফল 1 = 0;
জন্য (i=1; i<=n; i++)                    // এর জন্য প্রথম ডাবল লুপ (j=1; j<=n; j+
      +) // do n গুন sum1++;

যোগফল 2 = 0;
জন্য (i=1; i<=n; i++)                    // এর জন্য দ্বিতীয় ডবল লুপ (j=1; j<=i; j+
      +) // do i times sum2++;
```

প্রথম ডাবল লুপে, লুপের জন্য অভ্যন্তরীণ সবসময় n বার চালায়।
যেহেতু বাইরের লুপটি n বার চালায়, এটি স্পষ্ট হওয়া উচিত যে স্টেট-ment sum1++ সঠিকভাবে n বার চালানো হয়। দ্বিতীয় লুপটি খরচ সহ পূর্ববর্তী উদাহরণে বিশ্লেষণ করা অনুরূপ

$$\sum_{j=1}^{n} j$$ এটি হল ap-

প্রথমের প্রায় $\frac{1}{2}n^2$. সুতরাং, উভয় ডাবল লুপের দাম Θ(n ^{2}), যদিও দ্বিতীয় অর্ধেক সময় প্রয়োজন।

উদাহরণ 3.13 লুপগুলির জন্য সমস্ত দ্বিগুণ নেস্টেড নয় Θ(n ing জোড়া ^{2}) অনুসরণ-নেস্টেড লুপগুলি এই সত্যটিকে ব্যাখ্যা করে।

```
যোগফল 1 =
0; জন্য (k=1; k<=n; k*=2)                    // (j=1; j<=n; j++) এর জন্য n
      বার লগ করুন // n বার যোগফল1++ করুন;


যোগফল 2 = 0;
জন্য (j=1; j<=k; j++) এর জন্য n বার          // (k=1; k<=n; k*=2) এর
      লগ করুন // k বার যোগফল2++ করুন;
```

এই দুটি কোড খণ্ড বিশ্লেষণ করার সময়, আমরা অনুমান করব যে n দুটির একটি শক্তি। প্রথম কোড ফ্র্যাগমেন্টের বাইরের জন্য লুপ এক্সিকিউট করা লগ n + 1 বার রয়েছে কারণ প্রতিটি পুনরাবৃত্তিতে k কে দুই দ্বারা গুণ করা হয় যতক্ষণ না এটি n এ পৌঁছায়। কারণ অভ্যন্তরীণ লুপ সর্বদা n বার চালায়, প্রথম কোড খণ্ডের জন্য মোট খরচ প্রকাশ করা যেতে পারে কারণ এখানে যোগফল তৈরি করতে প্রতিস্থাপন করা হয়, k = 2i সমীকরণ 2.3 $\log n$ n. উল্লেখ্য যে একটি পরিবর্তনশীল $i=0$ সহ, এই যোগফলের সমাধান হল Θ(n log n)। দ্বিতীয় কোড ফ্র্যাগমেন্টে, বাইরের লুপটিও $\cdot$ থেকে লগ n + 1 বার কার্যকর করা হয়। ভিতরের লুপের খরচ k, যা প্রতিবার দ্বিগুণ হয়। যোগফলকে প্রকাশ করা যেতে পারে যেখানে n কে দুই এবং আবার k = 2i এর শক্তি বলে ধরে নেওয়া হয়

হিসাবে $\log n$ $i\ 2\ i=0$.

সমীকরণ 2.8 থেকে, আমরা জানি যে এই যোগফলটি কেবল Θ(n)।

অন্যান্য নিয়ন্ত্রণ বিবৃতি সম্পর্কে কি? যখন লুপগুলি লুপগুলির অনুরূপভাবে বিশ্লেষণ করা হয় । সবচেয়ে খারাপ ক্ষেত্রে একটি if স্টেটমেন্টের খরচ তারপর এবং অন্য ধারাগুলির খরচের চেয়ে বেশি । এটি গড় ক্ষেত্রের ক্ষেত্রেও সত্য, অনুমান করে যে n-এর আকার ধারাগুলির একটি কার্যকর করার সম্ভাবনাকে প্রভাবিত করে না (যা সাধারণত, তবে অগত্যা সত্য নয়)। সুইচ স্টেটমেন্টের জন্য , সবচেয়ে খারাপ ক্ষেত্রে খরচ সবচেয়ে ব্যয়বহুল শাখার। সাবরুটিন কলের জন্য, সাবরুটিন চালানোর খরচ যোগ করুন।

এমন বিরল পরিস্থিতি রয়েছে যেখানে একটি if বা সুইচ স্টেটমেন্টের বিভিন্ন শাখা কার্যকর করার সম্ভাবনা ইনপুট আকারের ফাংশন। উদাহরণের জন্য, n আকারের ইনপুটের জন্য, একটি if স্টেটমেন্টের ধারাটি সম্ভাব্যতা 1/n সহ কার্যকর করা যেতে পারে। একটি উদাহরণ হল একটি if স্টেটমেন্ট যা শুধুমাত্র n মানের ক্ষুদ্রতম মানের জন্য তারপরের ধারাটি কার্যকর করে। এই জাতীয় প্রোগ্রামগুলির জন্য একটি গড়-কেস বিশ্লেষণ সম্পাদন করার জন্য, আমরা আরও ব্যয়বহুল শাখার ব্যয় হিসাবে if স্টেটমেন্টের ব্যয়কে কেবল গণনা করতে পারি না। এই ধরনের পরিস্থিতিতে, পরিমার্জিত বিশ্লেষণের কৌশল (বিভাগ 14.3 দেখুন) উদ্ধারে আসতে পারে।

একটি পুনরাবৃত্ত সাবরুটিনের সম্পাদনের সময় নির্ধারণ করা কঠিন হতে পারে। একটি পুনরাবৃত্ত সাবরুটিনের জন্য চলমান সময় সাধারণত একটি পুনরাবৃত্তি সম্পর্ক দ্বারা সবচেয়ে ভালভাবে প্রকাশ করা হয়। উদাহরণ স্বরূপ, সেকশন 2.5-এর রিকার্সিভ ফ্যাক্টোরিয়াল ফাংশন ফ্যাক্ট নিজেকে এর ইনপুট মানের থেকে এক কম মান দিয়ে কল করে। এই রিকার্সিভ কলের ফলাফল

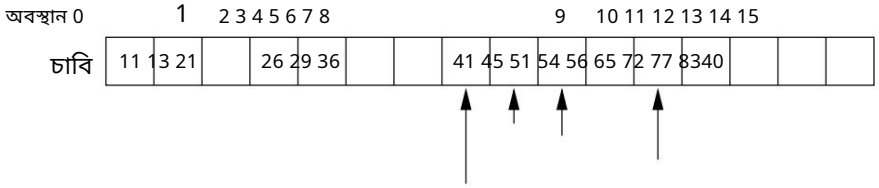
তারপর ইনপুট মান দ্বারা গুণিত হয়, যা ধ্রুবক সময় নেয়। এইভাবে, খরচ
ফ্যাক্টরিয়াল ফাংশন, যদি আমরা মাল্টি-প্লিকেশন ক্রিয়াকলাপের সংখ্যার পরিপ্রেক্ষিতে খরচ পরিমাপ করতে চাই, তাহলে এটি দ্বারা করা গুণের সংখ্যার চেয়ে এক বেশি।
ছোট ইনপুট উপর recursive কল. কারণ বেস কেস কোন গুন করে না,
এর খরচ শূন্য। সুতরাং, এই ফাংশনের জন্য চলমান সময় হিসাবে প্রকাশ করা যেতে পারে

$$T(n) = T(n - 1) + 1 \text{ এর জন্য } n > 1; T(1) = 0।$$

আমরা উদাহরণ 2.8 এবং 2.13 থেকে জানি যে এই পুনরাবৃত্ত-পুনরুক্তি সম্পর্কের জন্য বন্ধ-ফর্ম সমাধান হল Θ(n)।

এই বিভাগের জন্য অ্যালগরিদম বিশ্লেষণের চূড়ান্ত উদাহরণ একটি অ্যারেতে অনুসন্ধান করার জন্য দুটি অ্যালগো-রিদমের তুলনা করবে। এর আগে আমরা নির্ধারণ করেছিলাম যে চলমান
একটি অ্যারেতে অনুক্রমিক অনুসন্ধানের জন্য সময় যেখানে অনুসন্ধান মান K সমানভাবে সম্ভব
যেকোন স্থানে উপস্থিত হওয়া হল Θ(n) গড় এবং সবচেয়ে খারাপ উভয় ক্ষেত্রেই। আমরা করব
এই চলমান সময়ের সাথে একটি বাইনারি অনুসন্ধান চালানোর জন্য প্রয়োজনীয় সময়ের সাথে তুলনা করতে চাই
একটি অ্যারে যার মান সর্বনিম্ন থেকে সর্বোচ্চ পর্যন্ত সংরক্ষিত হয়।

বাইনারি অনুসন্ধান ar-ray এর মধ্যম অবস্থানে মান পরীক্ষা করে শুরু হয়; এই অবস্থানটিকে মধ্য এবং সংশ্লিষ্ট মানটিকে kmid বলুন। যদি kmid = K, তাহলে
প্রক্রিয়াকরণ অবিলম্বে বন্ধ হতে পারে। তবে এটি হওয়ার সম্ভাবনা কম। সৌভাগ্যবশত, মধ্যম মান জানা দরকারী তথ্য প্রদান করে যা গাইড করতে সাহায্য করতে পারে
অনুসন্ধান প্রক্রিয়া। বিশেষ করে, যদি kmid > K, তাহলে আপনি জানেন যে মান K
মধ্য থেকে বড় কোনো অবস্থানে অ্যারে উপস্থিত হতে পারে না. এইভাবে, আপনি অ্যারের উপরের অর্ধেকের ভবিষ্যতের অনুসন্ধানটি বাদ দিতে পারেন। বিপরীতভাবে, যদি kmid < K, তাহলে
আপনি জানেন যে আপনি মধ্য থেকে কম অ্যারের সমস্ত অবস্থান উপেক্ষা করতে পারেন। যেভাবেই হোক,
পদের অর্ধেক আরও বিবেচনা থেকে বাদ দেওয়া হয়। বাইনারি অনুসন্ধান পরবর্তী
অ্যারের সেই অংশে মাঝামাঝি অবস্থান দেখে যেখানে K মান থাকতে পারে। দ্য
এই অবস্থানে মান আবার আমাদের অবশিষ্ট অবস্থানের অর্ধেক অপসারণ করতে অনুমতি দেয়
বিবেচনা থেকে পছন্দসই মান পাওয়া না যাওয়া পর্যন্ত এই প্রক্রিয়াটি পুনরাবৃত্তি হয়,
অথবা অ্যারেতে কোনো অবস্থান অবশিষ্ট নেই যেখানে K মান থাকতে পারে।
চিত্র 3.4 বাইনারি অনুসন্ধান পদ্ধতি চিত্রিত করে। এখানে একটি C++ বাস্তবায়ন রয়েছে
বাইনারি অনুসন্ধান:



চিত্র 3.4 16টি অবস্থানের একটি সাজানো অ্যারেতে বাইনারি অনুসন্ধানের একটি চিত্র।

মান K = 45 সহ অবস্থানের জন্য একটি অনুসন্ধান বিবেচনা করুন। বাইনারি অনুসন্ধান প্রথমে 7 অবস্থানে মান পরীক্ষা করে। কারণ 41 < K, অ্যারেতে 7 এর নিচে যে কোনও অবস্থানে কাঙ্ক্ষিত মান উপস্থিত হতে পারে না। পরবর্তী, বাইনারি অনুসন্ধান অবস্থান 11 এ মান পরীক্ষা করে।

কারণ 56 > K, পছন্দসই মান (যদি এটি বিদ্যমান থাকে) অবশ্যই অবস্থান 7 এবং 11 এর মধ্যে হতে হবে। অবস্থান 9 এর পরে চেক করা হয়েছে। আবার, এর মানও অনেক বেশি। চূড়ান্ত অনুসন্ধানটি 8 অবস্থানে রয়েছে, যা পছন্দসই মান ধারণ করে। এইভাবে, ফাংশন বাইনারি অবস্থান 8 প্রদান করে। বিকল্পভাবে, K যদি 44 হয়, তাহলে রেকর্ড অ্যাক্সেসের একই সিরিজ তৈরি করা হবে। অবস্থান 8 চেক করার পরে, বাইনারি n এর একটি মান প্রদান করবে, যা নির্দেশ করে যে অনুসন্ধানটি ব্যর্থ হয়েছে।

```
// "K" মান সহ // আকার "n" এর সাজানো অ্যারে "A" এ একটি উপাদানের অবস্থান ফেরত দিন। যদি "কে" "A" তে না থাকে, তাহলে // মান "n" ফেরত দিন। int বাইনারি (int A[], int n, int K) {

    int l = -1; // l এবং r
                        অ্যারে সীমার বাইরে int r = n; যখন (l+1 != r) { // l এবং r মিলিত হলে থামুন

        int i = (l+r)/2; // অবশিষ্ট সাবয়ারের মাঝখানে পরীক্ষা করুন
        r = i; যদি (K == A[i]) রিটার্ন i; // এটি                  // বাম অর্ধেক যদি (K < A[i])
        পাওয়া গেছে যদি (K > A[i]) l = i;
                                                    // ডান অর্ধেক

    } ফেরত n; // অনুসন্ধান মান A তে নেই
}
```

সবচেয়ে খারাপ ক্ষেত্রে এই অ্যালগরিদমের খরচ খুঁজে বের করতে, আমরা চলমান সময়কে পুনরাবৃত্তি হিসাবে মডেল করতে পারি এবং তারপরে বন্ধ-ফর্ম সমাধানটি খুঁজে পেতে পারি। বাইনারিতে প্রতিটি পুনরাবৃত্ত কল অ্যারের আকারকে প্রায় অর্ধেক করে দেয়, তাই আমরা নিম্নরূপ সবচেয়ে খারাপ-কেস খরচ মডেল করতে পারি, সরলতার জন্য অনুমান করে যে n হল দুটির শক্তি।

$$T(n) = T(n/2) + 1 \text{ এর জন্য } n > 1;\ T(1) = 1।$$

যদি আমরা পুনরাবৃত্তি প্রসারিত করি, আমরা দেখতে পাই যে আমরা বেস কেসে পৌঁছানোর আগে শুধুমাত্র লগ n বার করতে পারি এবং প্রতিটি প্রসারণ খরচে একটি যোগ করে। সুতরাং, পুনরাবৃত্তির জন্য বন্ধ-ফর্ম সমাধান হল T(n) = log n।

ফাংশন বাইনারি K-এর (একক) উপস্থিতি খুঁজে বের করতে এবং এর অবস্থান ফেরানোর জন্য ডিজাইন করা হয়েছে। অ্যারেতে K উপস্থিত না হলে একটি বিশেষ মান প্রদান করা হয়। এই অ্যালগরিদমটি পরিবর্তন করা যেতে পারে যেমন পজিশন রিটার্ন করার মত বৈচিত্র বাস্তবায়ন করতে

অ্যারেতে K-এর প্রথম সংঘটনের যদি একাধিক ঘটনার অনুমতি দেওয়া হয়, এবং K অ্যারেতে না থাকলে K-এর থেকে কম সর্বশ্রেষ্ঠ মানের অবস্থান ফেরত দেয়।

বাইনারি অনুসন্ধানের সাথে অনুক্রমিক অনুসন্ধানের তুলনা করে, আমরা দেখতে পাই যে n যত বাড়তে থাকে, গড় এবং সবচেয়ে খারাপ ক্ষেত্রে Θ(n) ক্রমিক অনুসন্ধানের জন্য চলমান সময় দ্রুত বাইনারি অনুসন্ধানের জন্য Θ(log n) চলমান সময়ের চেয়ে অনেক বেশি হয়ে যায়। বিচ্ছিন্নভাবে নেওয়া, বাইনারি অনুসন্ধান অনুক্রমিক অনুসন্ধানের চেয়ে অনেক বেশি কার্যকর বলে মনে হয়। বাইনারি অনুসন্ধানের জন্য ধ্রুবক ফ্যাক্টর অনুক্রমিক অনুসন্ধানের চেয়ে বেশি হওয়া সত্ত্বেও এটি হচ্ছে, কারণ বাইনারি অনুসন্ধানে পরবর্তী অনুসন্ধান অবস্থানের জন্য গণনা করা বর্তমান অবস্থানকে বৃদ্ধি করার চেয়ে বেশি ব্যয়বহুল, যেমনটি অনুক্রমিক অনুসন্ধান করে।

তবে মনে রাখবেন যে ক্রমিক অনুসন্ধানের জন্য চলমান সময় মোটামুটিভাবে একই হবে তা নির্বিশেষে অ্যারের মানগুলি ক্রমানুসারে সংরক্ষণ করা হয়েছে কিনা। বিপরীতে, বাইনারি অনুসন্ধানের জন্য অ্যারের মানগুলি সর্বনিম্ন থেকে সর্বোচ্চ পর্যন্ত অর্ডার করা প্রয়োজন। যে প্রসঙ্গে বাইনারি অনুসন্ধান ব্যবহার করা হবে সেই প্রসঙ্গে ডি-পেন্ডিং, একটি সাজানো অ্যারের জন্য এই প্রয়োজনীয়তা একটি সম্পূর্ণ প্রোগ্রামের চলমান সময়ের জন্য ক্ষতিকারক হতে পারে, কারণ নতুন উপাদানগুলি সন্নিবেশ করার সময় বাছাইকৃত ক্রমে মান বজায় রাখার জন্য আরও বেশি খরচের প্রয়োজন হয় অ্যারের মধ্যে এটি অনুসন্ধানের সময় বাইনারি অনুসন্ধানের অ্যাডভান-টেজ এবং একটি সাজানো অ্যারে বজায় রাখার সাথে সম্পর্কিত অসুবিধাগুলির মধ্যে একটি ট্রেডঅফের একটি উদাহরণ। শুধুমাত্র সম্পূর্ণ সমস্যা সমাধানের প্রেক্ষাপটে আমরা জানতে পারি সুবিধা অসুবিধার চেয়ে বেশি কিনা।

3.6 সমস্যা বিশ্লেষণ

আপনি প্রায়শই একটি অ্যালগরিদম বিশ্লেষণ করতে "অ্যালগরিদম" বিশ্লেষণের কৌশলগুলি ব্যবহার করেন, বা একটি প্রোগ্রাম হিসাবে একটি অ্যালগরিদমের ইনস্ট্যান্টেশন। আপনি একটি সমস্যার খরচ বিশ্লেষণ করতে এই একই কৌশল ব্যবহার করতে পারেন। এটা বলা আপনার কাছে বোধগম্য হওয়া উচিত যে একটি সমস্যার জন্য উপরের সীমাটি সেরা অ্যালগরিদমের জন্য উপরের সীমার চেয়ে খারাপ হতে পারে না যা আমরা সেই সমস্যার জন্য জানি। কিন্তু একটি সমস্যা জন্য একটি নিম্ন সীমা দিতে মানে কি?

একটি প্রদত্ত সমস্যার জন্য কিছু অ্যালগরিদমের জন্য একটি প্রদত্ত আকার n এর সমস্ত ইনপুটগুলির খরচের একটি গ্রাফ বিবেচনা করুন। A কে সংজ্ঞায়িত করুন সমস্ত অ্যালগরিদমের সংগ্রহ যা সমস্যার সমাধান করে (তাত্ত্বিকভাবে, এই জাতীয় অ্যালগরিদমের অসীম সংখ্যা রয়েছে)। এখন, A-তে সমস্ত (অসীম অনেক) অ্যালগরিদমের জন্য সমস্ত গ্রাফের সংগ্রহ বিবেচনা করুন। সবচেয়ে খারাপ ক্ষেত্রে নিম্ন সীমা হল সমস্ত গ্রাফের সমস্ত সর্বোচ্চ বিন্দুর মধ্যে সর্বনিম্ন।

একটি অ্যালগরিদম (অথবা প্রোগ্রাম) Ω(f(n)) এ আছে তা দেখানোর চেয়ে এটি দেখানো অনেক সহজ যে একটি সমস্যা Ω(f(n)) এ রয়েছে। একটি সমস্যা Ω(f(n)) তে থাকার মানে হল যে প্রতিটি অ্যালগরিদম যা সমস্যার সমাধান করে Ω(f(n)) এ রয়েছে, এমনকি অ্যালগরিদম যা আমরা ভাবিনি!

এখনও পর্যন্ত আমাদের অ্যালগরিদম বিশ্লেষণের সমস্ত উদাহরণ "স্পষ্ট" ফলাফল দেয়, বড়-ওহ সর্বদা Ω মেলে। একটি সমস্যা বা অ্যালগরিদম সম্পর্কে আমাদের বোঝার বর্ণনা দিতে কত বড়-ওহ, Ω, এবং Θ স্বরলিপি সঠিকভাবে ব্যবহৃত হয় তা বোঝার জন্য, একটি উদাহরণ বিবেচনা করা ভাল যেখানে আপনি ইতিমধ্যে সমস্যা সম্পর্কে অনেক কিছু জানেন না।

এই প্রক্রিয়াটি কীভাবে কাজ করে তা দেখতে বাছাই করার সমস্যাটি বিশ্লেষণ করার জন্য এগিয়ে আসুন। সবচেয়ে খারাপ ক্ষেত্রে কোনো বাছাই অ্যালগরিদমের জন্য সর্বনিম্ন সম্ভাব্য খরচ কি?
অ্যালগরিদমকে অন্তত ইনপুটের প্রতিটি উপাদানের দিকে তাকাতে হবে, কেবলমাত্র ইনপুটটি সত্যই সাজানো হয়েছে তা নির্ধারণ করতে। এটাও সম্ভব যে প্রতিটি n মানকে অবশ্যই সাজানো আউটপুটে অন্য অবস্থানে সরানো হবে। এইভাবে, যেকোনো সাজানোর অ্যালগরিদমকে কমপক্ষে cn সময় নিতে হবে। অনেক সমস্যার জন্য, এই পর্যবেক্ষণ যে প্রতিটি n ইনপুট দেখতে হবে তা একটি সহজ Ω(n) নিম্ন সীমার দিকে নিয়ে যায়।

আপনার কম্পিউটার বিজ্ঞানের পূর্ববর্তী গবেষণায়, আপনি সম্ভবত একটি উদাহরণ দেখেছেন) সবচেয়ে একটি সাজানোর অ্যালগরিদম যার চলমান সময় O(n ^{2}খারাপ ক্ষেত্রে। সহজ
প্রথম বর্ষের প্রোগ্রামিং কোর্সে সাধারণত উদাহরণ হিসেবে দেওয়া বাবল সর্ট এবং ইনসার্শন সর্ট অ্যালগরিদমগুলি O(n-এ বাছাইয়ের ক্ষেত্রে সবচেয়ে খারাপ সময় থাকে যা বলা যেতে পারে O(n) ^{2}) এইভাবে, সমস্যা
তে উপরের বাউন্ড আছে। ^{2}) আমরা কিভাবে বন্ধ করব
Ω(n) এবং O(n-এর মধ্যে ব্যবধান ^{2})? একটি ভাল বাছাই অ্যালগরিদম হতে পারে? আপনি যদি পারেন), এমন কোনও অ্যালগরিদমের কথা ভাবেন না যার সবচেয়ে খারাপ ক্ষেত্রে বৃদ্ধির হার O(n) ^{2}এবং যদি আপনি এর চেয়ে ভাল তা দেখানোর জন্য কোনও বিশ্লেষণ কৌশল আবিষ্কার করেনি যে সবচেয়ে খারাপ ক্ষেত্রে বাছাই করার সমস্যার জন্য সর্বনিম্ন খরচ Ω-এর চেয়ে বেশি (n), তাহলে আপনি নিশ্চিতভাবে জানতে পারবেন না যে একটি ভাল অ্যালগরিদম আছে কি না।

অধ্যায় 7 বাছাই করার অ্যালগরিদম উপস্থাপন করে যার চলমান সময় সবচেয়ে খারাপ ক্ষেত্রে O(n log n) এ রয়েছে। এটি ব্যাপকভাবে ব্যবধানকে সংকুচিত করে। তার নতুন জ্ঞানের সাথে, আমাদের এখন Ω(n) এ একটি নিম্ন সীমা এবং O(n log n) এ একটি উপরের সীমা রয়েছে। আমরা একটি দ্রুত অ্যালগরিদম জন্য অনুসন্ধান করা উচিত? অনেকে চেষ্টা করেও সফল হয়নি। সৌভাগ্যবশত (বা সম্ভবত দুর্ভাগ্যবশত?), অধ্যায় 7-এ একটি প্রমাণও রয়েছে যে যেকোনো বাছাই অ্যালগরিদমের সবচেয়ে খারাপ ক্ষেত্রে Ω(n log n) এ চলমান সময় থাকতে হবে।2 এই প্রমাণটি অ্যালগরিদম বিশ্লেষণের ক্ষেত্রে সবচেয়ে গুরুত্বপূর্ণ ফলাফলগুলির মধ্যে একটি। , এবং এর মানে হল যে কোনও সাজানোর অ্যালগরিদম সম্ভবত n আকারের সবচেয়ে খারাপ-কেস ইনপুটের জন্য cn log n এর চেয়ে দ্রুত চলতে পারে না।
এইভাবে, আমরা উপসংহারে পৌঁছাতে পারি যে বাছাই করার সমস্যা হল Θ(n log n) সবচেয়ে খারাপ ক্ষেত্রে, কারণ উপরের এবং নীচের সীমাগুলি মিলিত হয়েছে।

একটি সমস্যার জন্য নিম্ন সীমা জানা আপনাকে একটি ভাল অ্যালগরিদম দেয় না।
কিন্তু কখন তাকানো বন্ধ করতে হবে তা জানতে এটি আপনাকে সাহায্য করে। যদি সমস্যার জন্য নিম্ন সীমাটি অ্যালগরিদমের (একটি ধ্রুবক ফ্যাক্টরের মধ্যে) উপরের সীমার সাথে মিলে যায়, তবে আমরা জানি যে আমরা একটি অ্যালগরিদম খুঁজে পেতে পারি যা শুধুমাত্র একটি ধ্রুবক গুণকের দ্বারা ভাল।

2যদিও সত্য জানা সৌভাগ্যের, তবে Θ(n) এর পরিবর্তে বাছাই করা Θ(n log n) হওয়া দুর্ভাগ্যজনক!

3.7 সাধারণ ভুল বোঝাবুঝি

অ্যাসিম্পোটিক বিশ্লেষণ হল সবচেয়ে বুদ্ধিবৃত্তিকভাবে কঠিন বিষয়গুলির মধ্যে একটি যা আন্ডারগ্র্যাড-ইউয়েট কম্পিউটার সায়েন্স মেজরদের মুখোমুখি হয়। বেশিরভাগ লোক বৃদ্ধির হার এবং অ্যাসিম্পোটিক বিশ্লেষণকে বিভ্রান্তিকর বলে মনে করে এবং তাই ধারণা বা পরিভাষা সম্পর্কে ভুল ধারণা তৈরি করে। এটি বিভ্রান্তির মানক পয়েন্টগুলি কী তা জানতে সাহায্য করে, সেগুলি এড়ানোর আশায়।

উপরের এবং নিম্ন সীমার ধারণাগুলিকে আলাদা করার ক্ষেত্রে একটি সমস্যা হল যে, বেশিরভাগ অ্যালগরিদমের জন্য আপনি যেগুলি সম্মুখীন হবেন, সেই অ্যালগরিদমের প্রকৃত বৃদ্ধির হার চিনতে সহজ। একটি খরচ ফাংশন সম্পর্কে সম্পূর্ণ জ্ঞান দেওয়া হলে, সেই খরচ ফাংশনের জন্য উপরের এবং নিম্ন সীমা সবসময় একই থাকে। সুতরাং, একটি উপরের এবং একটি নিম্ন সীমার মধ্যে পার্থক্য তখনই সার্থক হয় যখন আপনার পরিমাপ করা জিনিস সম্পর্কে অসম্পূর্ণ জ্ঞান থাকে। যদি এই পার্থক্যটি এখনও পরিষ্কার না হয়, তাহলে বিভাগ 3.6 পুনরায় পড়ুন। আমরা Θ-নোটেশন ব্যবহার করি ইঙ্গিত করার জন্য যে আমরা উপরের এবং নিম্ন সীমার বৃদ্ধির হার সম্পর্কে যা জানি তার মধ্যে কোন অর্থপূর্ণ পার্থক্য নেই (যা সাধারণত সাধারণ অ্যালগরিদমের ক্ষেত্রে হয়)।

একদিকে আপার বাউন্ড বা লোয়ার বাউন্ড এবং অন্যদিকে খারাপ কেস বা বেস্ট কেস ধারণাগুলিকে বিভ্রান্ত করা একটি সাধারণ ভুল। সর্বোত্তম, সবচেয়ে খারাপ বা গড় প্রতিটি ক্ষেত্রে আমাদের একটি নির্দিষ্ট উদাহরণ দেয় যা আমরা একটি খরচ পরিমাপ পেতে একটি অ্যালগরিদম বিবরণে প্রয়োগ করতে পারি। উপরের এবং নীচের সীমাগুলি সেই খরচ পরিমাপের জন্য বৃদ্ধির হার সম্পর্কে আমাদের বোঝার বর্ণনা দেয়। তাই একটি অ্যালগরিদম বা সমস্যার জন্য বৃদ্ধির হার সংজ্ঞায়িত করার জন্য, আমাদের নির্ধারণ করতে হবে আমরা কী পরিমাপ করছি (সর্বোত্তম, সবচেয়ে খারাপ, বা গড় ক্ষেত্রে) এবং সেই খরচ পরিমাপের বৃদ্ধির হার সম্পর্কে আমরা কী জানি তার জন্য আমাদের বিবরণ (বড়-ওহ , Ω, বা Θ)।

একটি অ্যালগরিদমের উপরের সীমাটি n আকারের প্রদত্ত ইনপুটের জন্য সেই অ্যালগরিদমের সবচেয়ে খারাপ ক্ষেত্রের মতো নয়। যা আবদ্ধ করা হচ্ছে তা প্রকৃত খরচ নয় (যা আপনি n এর একটি প্রদত্ত মানের জন্য নির্ধারণ করতে পারেন), বরং খরচের বৃদ্ধির হার। একটি একক বিন্দুর জন্য বৃদ্ধির হার হতে পারে না, যেমন n এর একটি নির্দিষ্ট মান। ইনপুট আকারে পরিবর্তন ঘটলে বৃদ্ধির হার খরচের পরিবর্তনের ক্ষেত্রে প্রযোজ্য।

একইভাবে, নিম্ন সীমা একটি প্রদত্ত আকার n এর জন্য সেরা ক্ষেত্রের মতো নয়।

আরেকটি সাধারণ ভ্রান্ত ধারণা হল যে একটি অ্যালগরিদমের জন্য সর্বোত্তম ক্ষেত্রে ঘটে যখন ইনপুট আকার যতটা সম্ভব ছোট হয়, বা সবচেয়ে খারাপ ঘটনা ঘটে যখন ইনপুট আকার যতটা সম্ভব বড় হয়। যেটি সঠিক তা হল প্রতিটি সম্ভাব্য আকারের ইনপুটের জন্য সেরা- এবং খারাপ-ক্ষেত্রের উদাহরণ বিদ্যমান। অর্থাৎ, একটি প্রদত্ত আকারের সমস্ত ইনপুটগুলির জন্য, বলুন, i আকারের ইনপুটগুলির মধ্যে একটি (বা একাধিক) সেরা এবং i আকারের ইনপুটগুলির মধ্যে একটি (বা একাধিক) সবচেয়ে খারাপ। প্রায়শই (কিন্তু সবসময় নয়!), আমরা একটি নির্বিচারে আকারের জন্য সেরা ইনপুট কেসটিকে চিহ্নিত করতে পারি এবং আমরা একটি নির্বিচারী আকারের জন্য সবচেয়ে খারাপ ইনপুট কেসটিকে চিহ্নিত করতে পারি। আদর্শভাবে, ইনপুট আকার বৃদ্ধির সাথে সাথে আমরা সেরা, সবচেয়ে খারাপ এবং গড় ক্ষেত্রে বৃদ্ধির হার নির্ধারণ করতে পারি।

উদাহরণ 3.14 অনুক্রমিক অনুসন্ধানের জন্য সেরা ক্ষেত্রে বৃদ্ধির হার কত? n আকারের যেকোন অ্যারের জন্য, সেরা ক্ষেত্রে ঘটে যখন আমরা যে মানটি খুঁজছি সেটি অ্যারের প্রথম অবস্থানে উপস্থিত হয়। এই অ্যারের আকার নির্বিশেষে সত্য. এইভাবে, সর্বোত্তম কেস (নিচ্ছাকৃত আকার n এর জন্য) ঘটে যখন পছন্দসই মানটি n অবস্থানের প্রথমটিতে থাকে এবং এর মূল্য 1 হয়। এটি বলা সঠিক নয় যে n = 1 হলে সর্বোত্তম ক্ষেত্রে ঘটে।

উদাহরণ 3.15 কল্পনা করুন যে n মানের মধ্যে সর্বাধিক মান খুঁজে পাওয়ার খরচ দেখানোর জন্য একটি গ্রাফ অঙ্কন করুন, n বৃদ্ধির সাথে সাথে। অর্থাৎ, x অক্ষ হবে n, এবং y মান হবে খরচ। অবশ্যই, এটি একটি তির্যক রেখা যা ডানদিকে যাচ্ছে, n বাড়ার সাথে সাথে (আপনি আরও পড়ার আগে এই গ্রাফটি নিজের জন্য স্কেচ করতে চাইতে পারেন)।

এখন, একটি অ্যারের মধ্যে (বলুন) 20টি উপাদানের মধ্যে সর্বাধিক মান খুঁজে পাওয়ার সমস্যা-লেমের প্রতিটি উদাহরণের জন্য খরচ দেখানো গ্রাফটি কল্পনা করুন।
গ্রাফের x অক্ষ বরাবর প্রথম অবস্থানটি অ্যারের প্রথম অবস্থানে সর্বাধিক উপাদান থাকার অনুরূপ হতে পারে। গ্রাফের x অক্ষ বরাবর দ্বিতীয় অবস্থানটি অ্যারের দ্বিতীয় অবস্থানে সর্বাধিক el-ement থাকার সাথে মিলে যেতে পারে এবং আরও অনেক কিছু। অবশ্যই, খরচ সর্বদা 20। অতএব, গ্রাফটি 20 মান সহ একটি অনুভূমিক রেখা হবে।

আপনার নিজের জন্য এই গ্রাফটি স্কেচ করা উচিত।

এখন, একটি অ্যারেতে একটি প্রদত্ত মানের জন্য একটি অনুক্রমিক অনুসন্ধান করার সমস্যার দিকে স্যুইচ করা যাক। সাইজ 20 এর সমস্ত সমস্যার উদাহরণ দেখানো গ্রাফটি সম্পর্কে চিন্তা করুন। প্রথম সমস্যাটি হতে পারে যখন আমরা যে মানটি অনুসন্ধান করি সেটি অ্যারের প্রথম অবস্থানে থাকে। এটির খরচ হয়েছে 1। দ্বিতীয় সমস্যাটি হতে পারে যখন আমরা যে মানটি অনুসন্ধান করি সেটি অ্যারের দ্বিতীয় অবস্থানে থাকে। এই খরচ হয়েছে 2. এবং তাই. যদি আমরা আকার 20 এর সমস্যা দৃষ্টান্তগুলিকে বাম দিকের সর্বনিম্ন ব্যয়বহুল থেকে ডানদিকে সবচেয়ে ব্যয়বহুল পর্যন্ত সাজাই, আমরা দেখতে পাই যে গ্রাফটি নীচের বাম থেকে (মান 0 সহ) উপরের ডানদিকে (মান 20 সহ) একটি তির্যক রেখা তৈরি করে। নিজের জন্য এই গ্রাফটি স্কেচ করুন।

অবশেষে, অ্যারে n এর আকার বড় হওয়ার সাথে সাথে অনুক্রমিক অনুসন্ধান সম্পাদনের জন্য খরচ বিবেচনা করা যাক। এই গ্রাফটি কেমন হবে? দুর্ভাগ্যবশত, একটি সহজ উত্তর নেই, যেমন সর্বাধিক মান খুঁজে বের করার জন্য ছিল। এই গ্রাফের আকৃতি নির্ভর করে আমরা সেরা কেস খরচ (এটি মান 1 সহ একটি অনুভূমিক রেখা হবে), সবচেয়ে খারাপ ক্ষেত্রে খরচ (এটি x অক্ষ বরাবর i অবস্থানে i মান সহ একটি তির্যক রেখা হবে) বিবেচনা করছি কিনা তার উপর নির্ভর করে। অথবা গড় খরচ

x অক্ষ বরাবর i অবস্থান)। এই কারণেই আমাদের সর্বদা বলতে হবে যে ফাংশন f(n) হল O(g(n)) সেরা, গড় বা সবচেয়ে খারাপ ক্ষেত্রে! যদি আমরা কোন শ্রেণীর ইনপুট নিয়ে আলোচনা করছি তা ছেড়ে দিলে, আমরা বেশিরভাগ অ্যালগরিদমের জন্য কোন খরচ পরিমাপের কথা বলছি তা আমরা জানতে পারি না।

3.8 একাধিক পরামিতি

কখনও কখনও একটি অ্যালগরিদমের সঠিক বিশ্লেষণের জন্য ব্যয় নির্ণয় করতে একাধিক প্যারামিটারের প্রয়োজন হয়। ধারণাটি ব্যাখ্যা করার জন্য, একটি ছবিতে সমস্ত পিক্সেল মানের গণনার জন্য র্যাঙ্ক ক্রম গণনা করার জন্য একটি অ্যালগরিদম বিবেচনা করুন। ছবি প্রায়ই একটি দ্বি-মাত্রিক অ্যারে দ্বারা প্রতিনিধিত্ব করা হয়, এবং একটি পিক্সেল অ্যারের একটি কক্ষ। একটি পিক্সেলের মান হল রঙের কোড মান, অথবা সেই পিক্সেলের ছবির তীব্রতার জন্য একটি মান। অনুমান করুন যে প্রতিটি পিক্সেল 0 থেকে C – 1 পরিসরে যেকোনো পূর্ণসংখ্যার মান নিতে পারে।
সমস্যাটি হল প্রতিটি রঙের মানের পিক্সেলের সংখ্যা খুঁজে বের করা এবং তারপরে প্রতিটি মান ছবিতে প্রদর্শিত সংখ্যার সাথে সাপেক্ষে রঙের মানগুলি সাজানো।
অনুমান করুন যে ছবিটি পি পিক্সেল সহ একটি আয়তক্ষেত্র। সমস্যা সমাধানের জন্য একটি সিউডোকোড অ্যালগরিদম অনুসরণ করে।

```
(i=0; i<C; i++) // গণনা গণনা শুরু করুন[i] = 0; (i=0; i<P; i++) // সমস্ত পিক্সেল দেখুন


    গণনা[মান(i)]++; // একটি পিক্সেল মান গণনা বৃদ্ধি করুন
                                        // বাছাই পিক্সেল মান গণনা সাজানোর (গণনা, সি);
```

এই উদাহরণে, গণনা হল সি আকারের একটি অ্যারে যা প্রতিটি রঙের মানের জন্য পিক্সেলের সংখ্যা সংরক্ষণ করে। ফাংশন মান(i) পিক্সেল i এর জন্য রঙের মান প্রদান করে।

প্রথম লুপের জন্য সময় (যা গণনা শুরু করে) রঙের সংখ্যার উপর ভিত্তি করে, C। দ্বিতীয় লুপের সময় (যা প্রতিটি রঙের সাথে পিক্সেলের সংখ্যা নির্ধারণ করে) হল Θ(P)। চূড়ান্ত লাইনের জন্য সময়, সাজানোর কল , ব্যবহৃত সাজানোর অ্যালগরিদমের খরচের উপর নির্ভর করে। অধ্যায় 3.6-এর আলোচনা থেকে, আমরা অনুমান করতে পারি যে সাজানোর অ্যালগরিদমের দাম Θ(P লগ P) আছে যদি P আইটেমগুলি সাজানো হয়, এইভাবে মোট অ্যালগরিদম খরচ হিসাবে Θ(P লগ P) পাওয়া যায়।

এই অ্যালগরিদম খরচ জন্য এটি একটি ভাল উপস্থাপনা? আসলে মিত্র বাছাই করা হচ্ছে কি? এটি পিক্সেল নয়, বরং রং। P থেকে C অনেক ছোট হলে কি হবে? তাহলে Θ(P log P) এর অনুমান হতাশাবাদী, কারণ P এর থেকে অনেক কম আইটেম সাজানো হচ্ছে। পরিবর্তে, প্রতিটি পিক্সেলের দিকে তাকানোর জন্য আমাদের বিশ্লেষণ ভেরিয়েবল হিসাবে P ব্যবহার করা উচিত এবং রঙগুলি দেখার পদক্ষেপগুলির জন্য আমাদের বিশ্লেষণ ভেরিয়েবল হিসাবে C ব্যবহার করা উচিত। তারপর আমরা ইনিশিয়ালাইজেশন লুপের জন্য Θ(C), পিক্সেল কাউন্ট লুপের জন্য Θ(P), এবং সাজানোর অপারেশনের জন্য Θ(C log C) পাব। এটি Θ(P + C log C) এর মোট খরচ দেয়।

কেন আমরা সহজভাবে ইনপুট আকারের জন্য C এর মান ব্যবহার করতে পারি না এবং বলতে পারি যে অ্যালগরিদমের খরচ হল Θ(C log C)? কারণ, C সাধারণত P থেকে অনেক কম। উদাহরণস্বরূপ, একটি ছবিতে 1000 × 1000 পিক্সেল এবং 256টি সম্ভাব্য রঙের পরিসর থাকতে পারে। সুতরাং, P হল এক মিলিয়ন, যা C লগ C থেকে অনেক বড়। কিন্তু, যদি P ছোট হয়, বা C বড় হয় (এমনকি যদি এটি P এর থেকেও কম হয়), তাহলে C লগ C বৃহত্তর পরিমাণে পরিণত হতে পারে। সুতরাং, কোন পরিবর্তনশীলকে উপেক্ষা করা উচিত নয়।

3.9 মহাকাশ সীমা

সময়ের পাশাপাশি, স্থান হল অন্যান্য কম্পিউটিং সংস্থান যা সাধারণত প্রোগ্রামারদের জন্য উদ্বেগের বিষয়। বছরের পর বছর ধরে কম্পিউটার যেমন দ্রুততর হয়েছে, তেমনি তারা মেমরিরও বেশি বরাদ্দ পেয়েছে। তবুও, উপলব্ধ ডিস্ক স্থান বা প্রধান মেমরির পরিমাণ অ্যালগরিদম ডিজাইনারদের জন্য উল্লেখযোগ্য সীমাবদ্ধতা হতে পারে।

স্থানের প্রয়োজনীয়তা পরিমাপ করতে ব্যবহৃত বিশ্লেষণ কৌশলগুলি সময়ের প্রয়োজনীয়তা পরিমাপের জন্য ব্যবহৃত হয়। যাইহোক, সময়ের প্রয়োজনীয়তাগুলি একটি অ্যালগরিদমের জন্য পরিমাপ করা হয় না যা একটি নির্দিষ্ট ডেটা কাঠামোকে ম্যানিপুলেট করে, স্থানের প্রয়োজনীয়তাগুলি সাধারণত ডেটা কাঠামোর জন্যই নির্ধারিত হয়। ইনপুট আকারে বৃদ্ধির হারের জন্য অ্যাসিম্পোটিক বিশ্লেষণের ধারণাগুলি স্থানের প্রয়োজনীয়তা পরিমাপের ক্ষেত্রে সম্পূর্ণরূপে প্রযোজ্য।

উদাহরণ 3.16 n inte-gers এর অ্যারের জন্য স্থানের প্রয়োজনীয়তাগুলি কী কী? যদি প্রতিটি পূর্ণসংখ্যার জন্য c বাইটের প্রয়োজন হয়, তাহলে অ্যারের জন্য cn বাইট প্রয়োজন, যা Θ(n)।

উদাহরণ 3.17 কল্পনা করুন যে আমরা n লোকেদের মধ্যে বন্ধুত্বের ট্র্যাক রাখতে চাই। আমরা n × n আকারের অ্যারে দিয়ে এটি করতে পারি। অ্যারের প্রতিটি সারি একজন ব্যক্তির বন্ধুদের প্রতিনিধিত্ব করে, কলামগুলি নির্দেশ করে যে সেই ব্যক্তিটি কার বন্ধু হিসাবে রয়েছে। উদাহরণস্বরূপ, যদি ব্যক্তি j ব্যক্তি i এর বন্ধু হয়, তাহলে আমরা অ্যারের i সারির j কলামে একটি চিহ্ন রাখি। একইভাবে, যদি আমরা অনুমান করি যে বন্ধুত্ব উভয় উপায়ে কাজ করে তবে আমাদের সারি j এর কলাম i এ একটি চিহ্ন স্থাপন করা উচিত। n লোকেদের জন্য, অ্যারের মোট আকার হল Θ(n ^{2})

একটি ডেটা স্ট্রাকচারের প্রাথমিক উদ্দেশ্য হল ডেটা এমনভাবে সংরক্ষণ করা যা সেই ডেটাগুলিতে দক্ষ অ্যাক্সেসের অনুমতি দেয়। দক্ষ অ্যাক্সেস প্রদানের জন্য, ডেটা কাঠামোর মধ্যে ডেটা কোথায় রয়েছে সে সম্পর্কে অতিরিক্ত তথ্য সংরক্ষণ করার প্রয়োজন হতে পারে। উদাহরণস্বরূপ, একটি লিঙ্ক করা তালিকার প্রতিটি নোডকে তালিকার পরবর্তী মানের জন্য একটি পয়েন্টার সংরক্ষণ করতে হবে। প্রকৃত ডেটা মান ছাড়াও সংরক্ষিত এই ধরনের সমস্ত তথ্যকে ওভারহেড হিসাবে উল্লেখ করা হয়।

আদর্শভাবে, সর্বোচ্চ অ্যাক্সেসের অনুমতি দেওয়ার সময় ওভারহেড ন্যূনতম রাখা উচিত।

এই বিরোধী লক্ষ্যগুলির মধ্যে একটি ভারসাম্য বজায় রাখার প্রয়োজনীয়তাই ডেটা স্ট্রাকচারের অধ্যয়নকে এত আকর্ষণীয় করে তোলে।

অ্যালগরিদম ডিজাইনের একটি গুরুত্বপূর্ণ দিককে স্থান/সময় ট্রেড-অফ নীতি হিসাবে উল্লেখ করা হয়। স্পেস/টাইম ট্রেডঅফ নীতি বলে যে কেউ প্রায়শই সময়ের হ্রাস অর্জন করতে পারে যদি কেউ স্থান ত্যাগ করতে ইচ্ছুক হয় বা এর বিপরীতে। অনেক প্রোগ্রাম "প্যাকিং" বা তথ্য এনকোডিং দ্বারা স্টোরেজ প্রয়োজনীয়তা কমাতে পরিবর্তন করা যেতে পারে। তথ্য "আনপ্যাক করা" বা ডিকোড করার জন্য অতিরিক্ত সময়ের প্রয়োজন। সুতরাং, ফলস্বরূপ প্রোগ্রামটি কম স্থান ব্যবহার করে তবে ধীর গতিতে চলে। বিপরীতভাবে, অনেক প্রোগ্রামগুলিকে প্রাক-স্টোর ফলাফলে পরিবর্তন করা যেতে পারে বা বৃহত্তর স্টোরেজ প্রয়োজনীয়তার খরচে দ্রুত চলমান সময়কে অনুমতি দেওয়ার জন্য তথ্য পুনর্গঠন করা যেতে পারে। সাধারণত, সময় এবং স্থান এই ধরনের পরিবর্তন একটি ধ্রুবক ফ্যাক্টর দ্বারা হয়.

স্পেস/টাইম ট্রেডঅফের একটি ক্লাসিক উদাহরণ হল লুকআপ টেবিল। একটি লুকআপ টেবিল একটি ফাংশনের মান প্রাক-সঞ্চয় করে যা অন্যথায় প্রতিবার প্রয়োজন হলে গণনা করা হবে। উদাহরণস্বরূপ, 12! ফ্যাক্টরিয়াল ফাংশনের জন্য সবচেয়ে বড় মান যা একটি 32-বিট int ভেরিয়েবলে সংরক্ষণ করা যেতে পারে। আপনি যদি এমন একটি প্রোগ্রাম লিখছেন যা প্রায়শই ফ্যাক্টরিয়ালগুলি গণনা করে, তাহলে একটি টেবিলে 12টি সংরক্ষণযোগ্য মানগুলি প্রাক-গণনা করার জন্য এটি অনেক বেশি সময় দক্ষ হতে পারে। যখনই প্রোগ্রামের n এর মান প্রয়োজন! n 12-এর জন্য, এটি কেবল অনুসন্ধান টেবিলটি পরীক্ষা করতে পারে। (যদি n > 12, মানটি যেভাবেই হোক একটি int ভেরিয়েবল হিসাবে সংরক্ষণ করার জন্য খুব বড়।) ফ্যাক্টরিয়াল গণনা করার জন্য প্রয়োজনীয় সময়ের তুলনায়, এটি লুকআপ টেবিল সংরক্ষণ করার জন্য প্রয়োজনীয় অতিরিক্ত স্থানের সামান্য পরিমাণের মূল্য হতে পারে।

লুকআপ টেবিলগুলি সাইন বা কোসাইনের মতো ব্যয়বহুল ফাংশনের জন্য অনুমান সংরক্ষণ করতে পারে। আপনি যদি এই ফাংশনটি শুধুমাত্র সঠিক ডিগ্রীর জন্য গণনা করেন বা নিকটতম ডিগ্রীর মান সহ উত্তরটি আনুমানিক করতে ইচ্ছুক হন, তাহলে সাইন ফাংশনটি বারবার গণনা করার পরিবর্তে সঠিক ডিগ্রীর জন্য গণনা সংরক্ষণকারী একটি লুকআপ টেবিল ব্যবহার করা যেতে পারে। লক্ষ্য করুন যে প্রাথমিকভাবে লুকআপ টেবিল তৈরি করতে একটি নির্দিষ্ট সময় প্রয়োজন। এই প্রারম্ভিকতাকে সার্থক করার জন্য আপনার অ্যাপ্লিকেশনটিকে অবশ্যই লুকআপ টেবিল ব্যবহার করতে হবে।

স্থান/সময় ট্রেডঅফের আরেকটি উদাহরণ হল স্থান অপ্টিমাইজ করার চেষ্টা করার সময় একজন প্রোগ্রামার কী সম্মুখীন হতে পারে। পূর্ণসংখ্যার একটি অ্যারে সাজানোর জন্য এখানে একটি সাধারণ কোড খণ্ড রয়েছে। আমরা অনুমান করি যে এটি একটি বিশেষ ক্ষেত্রে যেখানে n পূর্ণসংখ্যা রয়েছে যার মানগুলি 0 থেকে n −1 পর্যন্ত পূর্ণসংখ্যাগুলির একটি স্থানান্তর। এটি একটি বিনসর্টের উদাহরণ, যা 7.7 অনুচ্ছেদে আলোচনা করা হয়েছে। Binsort প্রতিটি মানকে তার মানের সাথে সম্পর্কিত একটি অ্যারের অবস্থানে বরাদ্দ করে।

```
জন্য (i=0; i<n; i++)
    B[A[i]] = A[i];
```

এটি কার্যকর এবং Θ(n) সময় প্রয়োজন। যাইহোক, এটি n আকারের দুটি অ্যারে প্রয়োজন। এর পরের একটি কোড খণ্ড যা ক্রমাগত ক্রমানুসারে রাখে কিন্তু একই অ্যারের মধ্যে এটি করে (এইভাবে এটি একটি "স্থানে" সাজানোর উদাহরণ)।

```
জন্য (i=0; i<n; i++)
    যখন (A[i]!= i)
        অদলবদল(A, i, A[i]);
```

ফাংশন অদলবদল(A, i, j) অ্যারে এ এলিমেন্ট i এবং j বিনিময় করে (পরিশিষ্ট দেখুন)। এটা স্পষ্ট নাও হতে পারে যে দ্বিতীয় কোড ফ্র্যাগমেন্ট আসলে অ্যারে সাজায়। এটি কাজ করে কিনা তা দেখতে, লক্ষ্য করুন যে লুপের মধ্য দিয়ে প্রতিটি পাস কমপক্ষে i মান সহ পূর্ণসংখ্যাকে অ্যারের সঠিক অবস্থানে নিয়ে যাবে এবং এই পুনরাবৃত্তির সময়, A[i] এর মান অবশ্যই বা এর চেয়ে বেশি হতে হবে i এর সমান। মোট সর্বাধিক n অদলবদল ক্রিয়াকলাপ সংঘটিত হয়, কারণ একটি পূর্ণসংখ্যা সেখানে স্থাপন করার পরে তার সঠিক অবস্থান থেকে সরানো যায় না এবং প্রতিটি সোয়াপ অপারেশন তার সঠিক অবস্থানে কমপক্ষে একটি পূর্ণসংখ্যা রাখে। এইভাবে, এই কোড খণ্ডের খরচ হয়েছে Θ(n)।

যাইহোক, প্রথম কোড খণ্ডের চেয়ে এটি চালানোর জন্য আরও বেশি সময় প্রয়োজন। আমার কম্পিউটারে দ্বিতীয় সংস্করণটি প্রথমটির তুলনায় প্রায় দ্বিগুণ সময় নেয়, তবে এটির জন্য কেবল অর্ধেক স্থান প্রয়োজন।

একটি প্রোগ্রামের স্থান এবং সময়ের প্রয়োজনীয়তার মধ্যে সম্পর্কের জন্য দ্বিতীয় নীতিটি সেই প্রোগ্রামগুলিতে প্রযোজ্য যা ডিস্কে সংরক্ষিত তথ্য প্রক্রিয়া করে, যেমনটি অধ্যায় 8 এবং তার পরে আলোচনা করা হয়েছে। আশ্চর্যজনকভাবে, ডিস্ক-ভিত্তিক স্থান/সময় ট্রেডঅফ নীতিটি প্রধান মেমরি ব্যবহার করে প্রোগ্রামের জন্য স্থান/সময় ট্রেডঅফ নীতির প্রায় বিপরীত।

ডিস্ক -ভিত্তিক স্থান/সময় ট্রেডঅফ নীতি বলে যে আপনি আপনার ডিস্ক স্টোরেজ প্রয়োজনীয়তা যত কম করতে পারবেন, আপনার প্রোগ্রাম তত দ্রুত চলবে। এটি এই কারণে যে ডিস্ক থেকে তথ্য পড়ার সময় গণনার সময়ের তুলনায় প্রচুর, তাই ডেটা আনপ্যাক করার জন্য প্রয়োজনীয় প্রায় কোনও অতিরিক্ত গণনা স্টোরেজ প্রয়োজনীয়তা হ্রাস করে ডিস্ক-পড়ার সময়ের চেয়ে কম হতে চলেছে। . স্বাভাবিকভাবেই এই নীতিটি সব ক্ষেত্রেই সত্য নয়, তবে ডিস্কে সংরক্ষিত তথ্য প্রক্রিয়াকরণ প্রোগ্রাম ডিজাইন করার সময় এটি মনে রাখা ভাল।

3.10 আপনার প্রোগ্রামের গতি বাড়ান

অনুশীলনে, একটি অ্যালগরিদম যার বৃদ্ধির হার হল Θ(n) এবং আরেকটি যার বৃদ্ধির হার হল Θ(n log n) এর মধ্যে চলমান সময়ের মধ্যে এত বড় পার্থক্য নেই। তবে, Θ(n log n) এবং Θ(n সাধারণ ডেটা স্ট্রাকচার এবং অ্যালগরিদমের বৃদ্ধির হার সহ অ্যালগরিদমগুলির মধ্যে চলমান সময়ের মধ্যে একটি বিশাল পার্থক্য রয়েছে, এটি অস্বাভাবিক নয় যে একটি সমস্যা ^{2}) যেমনটি আপনি আপনার অধ্যয়নের সময় দেখতে পাবেন যার সুস্পষ্ট সমাধানের জন্য Θ(n) প্রয়োজন

^{2}) সময়েরও একটি সমাধান আছে যার জন্য প্রয়োজন Θ(n log n)

সময় উদাহরণগুলির মধ্যে রয়েছে বাছাই করা এবং অনুসন্ধান করা, দুটি সবচেয়ে গুরুত্বপূর্ণ কম্পিউটার সমস্যা।

উদাহরণ 3.18 নিচের একটি সত্য ঘটনা। কয়েক বছর আগে, আমার স্নাতক ছাত্রদের মধ্যে একটি বড় সমস্যা ছিল. তার থিসিস কাজ একটি বৃহৎ ডাটাবেস উপর বেশ কিছু জটিল অপারেশন জড়িত. তিনি এখন চূড়ান্ত ধাপে কাজ করছিলেন। "ডাঃ। শ্যাফার, "তিনি বলেছিলেন, "আমি এই প্রোগ্রামটি চালাচ্ছি এবং এটি দীর্ঘ সময় নিচ্ছে বলে মনে হচ্ছে।" অ্যালগরিদম পরীক্ষা করার পর আমরা বুঝতে পেরেছিলাম যে এটির চলমান সময় ছিল Θ(n সম্পূর্ণ করার জন্য। এমনকি যদি আমরা কম্পিউটারটিকে সেই দীর্ঘ সময়ের জন্য নিরবচ্ছিন্নভাবে চালিয়ে যেতে পারি, তবে তিনি তার থিসিসটি সম্পূর্ণ করার এবং তার আগেই স্নাতক হওয়ার আশা করেছিলেন। সৌভাগ্যবশত, আমরা ^{2}), এবং এটি সম্ভবত এক থেকে দুই সপ্তাহ সময় নেবে
বুঝতে পেরেছিলাম যে সেখানে একটি অ্যালগরিদমকে রূপান্তর করার জন্য মোটামুটি সহজ উপায় যাতে এটির চলমান সময় ছিল Θ(n log n)।

একটি অ্যালগরিদম পরিবর্তন করার মতো গুরুত্বপূর্ণ না হলেও, "কোড টিউনিং" চলমান সময়ে নাটকীয় উন্নতি ঘটাতে পারে। কোড টিউনিং হল একটি প্রোগ্রামকে দ্রুত চালানোর জন্য বা কম সঞ্চয়স্থানের প্রয়োজনের জন্য হ্যান্ড-অপ্টিমাইজ করার শিল্প।
অনেক প্রোগ্রামের জন্য, কোড টিউনিং চলমান সময়কে দশের ফ্যাক্টর দ্বারা কমাতে পারে, অথবা দুই বা তার বেশি ফ্যাক্টর দ্বারা স্টোরেজ প্রয়োজনীয়তা কমাতে পারে। আমি একবার একটি প্রোগ্রামে একটি সমালোচনামূলক ফাংশন টিউন করেছিলাম — এর মৌলিক অ্যালগরিদম পরিবর্তন না করেই — 200 স্পিডআপের একটি ফ্যাক্টর অর্জন করতে। এই গতি পাওয়ার জন্য, তবে, আমি তথ্যের উপস্থাপনায় বড় পরিবর্তন করেছি, একটি প্রতীকী কোডিং স্কিম থেকে একটি সংখ্যাসূচক কোডিং স্কিমে রূপান্তর করেছি যার উপর আমি সরাসরি গণনা করতে সক্ষম হয়েছি।

কোড টিউনিংয়ের মাধ্যমে আপনার প্রোগ্রামগুলিকে গতি বাড়ানোর উপায়গুলির জন্য এখানে কিছু পরামর্শ রয়েছে।
উপলব্ধি করা সবচেয়ে গুরুত্বপূর্ণ বিষয় হল যে একটি প্রোগ্রামের বেশিরভাগ বিবৃতি সেই প্রোগ্রামের চলমান সময়ের উপর খুব বেশি প্রভাব ফেলে না। সাধারণত কয়েকটি কী সাবরুটিন থাকে, সম্ভবত মূল সাবরুটিনের মধ্যে কোডের মূল লাইনও থাকে, যেটি বেশিরভাগ চলমান সময়ের জন্য অ্যাকাউন্ট। একটি সাবরুটিনের চলমান সময়ের অর্ধেক কাটাতে সামান্য কিছু নেই যা মোট চলমান সময়ের মাত্র 1%।

প্রোগ্রামের সেই অংশগুলিতে আপনার মনোযোগ কেন্দ্রীভূত করুন যা সবচেয়ে বেশি প্রভাব ফেলে।

কোড টিউন করার সময়, ভাল সময়ের পরিসংখ্যান সংগ্রহ করা গুরুত্বপূর্ণ। অনেক কম-পাইলার এবং অপারেটিং সিস্টেমের মধ্যে প্রোফাইলার এবং অন্যান্য বিশেষ সরঞ্জাম রয়েছে যা সময় এবং স্থান উভয়ের ব্যবহারের তথ্য সংগ্রহ করতে সহায়তা করে। একটি প্রোগ্রাম আরও দক্ষ করার চেষ্টা করার সময় এগুলি অমূল্য, কারণ তারা আপনাকে বলতে পারে আপনার প্রচেষ্টা কোথায় বিনিয়োগ করতে হবে।

অনেক কোড টিউনিং কাজের গতি বাড়ানোর পরিবর্তে কাজ এড়ানোর নীতির উপর ভিত্তি করে। একটি সাধারণ পরিস্থিতি ঘটে যখন আমরা একটি শর্তের জন্য পরীক্ষা করতে পারি

যা আমাদের কিছু কাজ এড়িয়ে যেতে দেয়। যাইহোক, এই ধরনের পরীক্ষা সম্পূর্ণ বিনামূল্যে হয় না। পরীক্ষার খরচ যাতে সংরক্ষিত কাজের পরিমাণের বেশি না হয় সেদিকে খেয়াল রাখতে হবে।

যদিও একটি পরীক্ষা সম্ভাব্যভাবে সংরক্ষিত কাজের চেয়ে সস্তা হতে পারে, পরীক্ষাটি সর্বদা করা উচিত এবং কাজটি সময়ের কিছু অংশ এড়ানো যেতে পারে।

উদাহরণ 3.19 কম্পিউটার গ্রাফিক্স অ্যাপ্লিকেশনের একটি সাধারণ কাজ হল জটিল বস্তুর সেটের মধ্যে কোনটি স্থানের একটি নির্দিষ্ট বিন্দু রয়েছে তা খুঁজে বের করা। এই সমস্যার বিভিন্নতা মোকাবেলা করার জন্য অনেক দরকারী ডেটা স্ট্রাকচার এবং অ্যালগরিদম তৈরি করা হয়েছে। এই ধরনের অধিকাংশ বাস্তবায়ন নিম্নলিখিত টিউনিং পদক্ষেপ জড়িত. প্রদত্ত জটিল অবজেক্টে প্রশ্নে বিন্দু রয়েছে কিনা তা সরাসরি পরীক্ষা করা তুলনামূলকভাবে ব্যয়বহুল। পরিবর্তে, বস্তুটির জন্য একটি বাউন্ডিং বাক্সের মধ্যে বিন্দুটি রয়েছে কিনা তা আমরা স্ক্রীন করতে পারি । বাউন্ডিং বক্সটি হল ক্ষুদ্রতম আয়তক্ষেত্র (সাধারণত x এবং y অক্ষের পাশে লম্বভাবে সংজ্ঞায়িত করা হয়) যাতে বস্তুটি থাকে।

যদি বিন্দুটি বাউন্ডিং বাক্সে না থাকে তবে এটি বস্তুতে থাকতে পারে না। যদি বিন্দুটি বাউন্ডিং বাক্সে থাকে, তবেই আমরা বিন্দু বনাম অবজেক্টের সম্পূর্ণ কম্প্যারিসন পরিচালনা করব। লক্ষ্য করুন যে যদি বিন্দুটি বাউন্ডিং বাক্সের বাইরে থাকে, তাহলে আমরা সময় বাঁচিয়েছি কারণ বাউন্ডিং বক্স পরীক্ষা সম্পূর্ণ বস্তু বনাম বিন্দুর তুলনার তুলনায় সস্তা। কিন্তু যদি বিন্দুটি বাউন্ডিং বাক্সের ভিতরে থাকে, তবে সেই পরীক্ষাটি অপ্রয়োজনীয় কারণ আমাদের এখনও অবজেক্টের বিপরীতে বিন্দুটিকে তুলনা করতে হবে। সাধারণত এই পরীক্ষাটি তৈরি করে যে পরিমান কাজ এড়ানো যায় তা প্রতিটি বস্তুর উপর পরীক্ষা করার খরচের চেয়ে বেশি।

উদাহরণ 3.20 বিভাগ 7.2.3 একটি বাছাই অ্যালগরিদম উপস্থাপন করে যার নাম সিলেকশন-শন সর্ট। এই অ্যালগরিদমের প্রধান স্বতন্ত্র বৈশিষ্ট্য হল যে এটি সাজানোর জন্য অ্যারেতে সংরক্ষিত রেকর্ডের অপেক্ষাকৃত কম অদলবদল প্রয়োজন।
যাইহোক, এটি কখনও কখনও একটি অপ্রয়োজনীয় সোয়াপ অপারেশন সঞ্চালন করে, নির্দিষ্টভাবে যখন এটি নিজের সাথে একটি রেকর্ড অদলবদল করার চেষ্টা করে। অদলবদল করা দুটি সূচক একই কিনা তা পরীক্ষা করে এই কাজটি এড়ানো যেতে পারে। যাইহোক, এই ঘটনা ঘটার অপর্যাপ্ত সম্ভাবনা আছে. যেহেতু পরীক্ষা সফল হওয়ার সময় সংরক্ষিত কাজের তুলনায় পরীক্ষার খরচ যথেষ্ট বেশি, তাই পরীক্ষাটি সাধারণত যোগ করার ফলে প্রোগ্রামটি গতির পরিবর্তে ধীর হয়ে যায়।

আপ

প্রোগ্রামটি অপঠনযোগ্য করে তোলে এমন কৌশলগুলি ব্যবহার না করার বিষয়ে সতর্ক থাকুন। বেশিরভাগ কোড টিউন-ইং হল একটি অযত্নে লিখিত প্রোগ্রাম পরিষ্কার করা, একটি পরিষ্কার প্রোগ্রাম গ্রহণ না করে এবং কৌশল যোগ করা। বিশেষ করে, আপনার ক্যাপা-এর জন্য একটি উপলব্ধি তৈরি করা উচিত-

এক্সপ্রেশনের অত্যন্ত ভাল অপ্টিমাইজেশন করতে আধুনিক কম্পাইলারগুলির বিলিটি।
"অভিব্যক্তির অপ্টিমাইজেশন" এর অর্থ হল আরও দক্ষতার সাথে চালানোর জন্য গাণিতিক বা যৌক্তিক অভিব্যক্তির পুনর্বিন্যাস। নিজের অভিব্যক্তিটিকে অপ্টিমাইজ করার প্রয়াসে আপনার জন্য এই ধরনের অপ্টিমাইজেশন করার জন্য কম্পাইলারের ক্ষমতার ক্ষতি না করার বিষয়ে সতর্ক থাকুন।
সর্বদা পরীক্ষা করুন যে আপনার "অপ্টিমাইজেশানগুলি" ইনপুটের একটি উপযুক্ত বেঞ্চমার্ক সেটে পরিবর্তনের আগে এবং পরে প্রোগ্রামটি চালানোর মাধ্যমে প্রোগ্রামটিকে সত্যিই উন্নত করে। আমার নিজের প্রো-গ্রামে কোড টিউনিংয়ের ইতিবাচক প্রভাব সম্পর্কে আমি অনেকবার ভুল করেছি। প্রায়শই আমি ভুল করি যখন আমি একটি অভিব্যক্তি অপ্টিমাইজ করার চেষ্টা করি। কম্পাইলারের চেয়ে ভাল করা কঠিন।

সর্বশ্রেষ্ঠ সময় এবং স্থান উন্নতি একটি ভাল ডেটা কাঠামো বা অ্যালগরিদম থেকে আসে। এই বিভাগের জন্য চূড়ান্ত চিন্তা

প্রথমে অ্যালগরিদম টিউন করুন, তারপর কোড টিউন করুন।

3.11 অভিজ্ঞতামূলক বিশ্লেষণ

এই অধ্যায়ে অ্যাসিম্পোটিক বিশ্লেষণের উপর দৃষ্টি নিবদ্ধ করা হয়েছে। এটি একটি বিশ্লেষণাত্মক টুল, যেখানে ইনপুট আকার বৃদ্ধির সাথে সাথে অ্যালগ-অরিদমের বৃদ্ধির হার নির্ধারণ করতে আমরা একটি অ্যালগরিদমের মূল দিকগুলি মডেল করি। পূর্বে উল্লেখ করা হয়েছে, এই পদ্ধতির অনেক সীমাবদ্ধতা আছে। এর মধ্যে রয়েছে ছোট সমস্যা আকারের প্রভাব, একই বৃদ্ধির হার সহ অ্যালগরিদমের মধ্যে সূক্ষ্ম পার্থক্য নির্ধারণ এবং আরও জটিল সমস্যার জন্য গাণিতিক মডেলিং করার অন্তর্নিহিত অসুবিধা।

বিশ্লেষণাত্মক পদ্ধতির একটি বিকল্প হল অভিজ্ঞতামূলক পদ্ধতি। সবচেয়ে স্পষ্ট অভিজ্ঞতামূলক পদ্ধতি হল দুটি প্রতিযোগীকে চালানো এবং কোনটি ভাল পারফর্ম করে তা দেখা। এইভাবে আমরা বিশ্লেষণাত্মক পদ্ধতির ঘাটতিগুলি কাটিয়ে উঠতে পারি।

সতর্ক থাকুন যে প্রোগ্রামগুলির তুলনামূলক সময় একটি কঠিন ব্যবসা, প্রায়শই অনিয়ন্ত্রিত কারণগুলির (সিস্টেম লোড, ব্যবহৃত ভাষা বা কম্পাইলার ইত্যাদি) থেকে উদ্ভূত পরীক্ষামূলক ত্রুটির সাপেক্ষে। সবচেয়ে গুরুত্বপূর্ণ বিষয় হল প্রোগ্রামগুলির একটির পক্ষে পক্ষপাতিত্ব না করা। আপনি যদি পক্ষপাতদুষ্ট হন তবে এটি নির্দিষ্ট সময়ে প্রতিফলিত হবে। প্রতিযোগী সফ্টওয়্যার বা হার্ডওয়্যার বিক্রেতাদের বিজ্ঞাপনের দিকে একবার নজর দিলে এটি আপনাকে বিশ্বাস করা উচিত। তাদের কর্মক্ষমতা তুলনা করার জন্য দুটি প্রোগ্রাম লেখার সময় সবচেয়ে সাধারণ সমস্যা হল যে একটি অন্যটির চেয়ে বেশি কোড-টিউনিং প্রচেষ্টা গ্রহণ করে। বিভাগ 3.10-এ উল্লিখিত হিসাবে, কোড টিউনিং প্রায়শই চলমান সময়কে দশের ফ্যাক্টর দ্বারা কমাতে পারে। যদি ইনপুট আকার নির্বিশেষে দুটি প্রোগ্রামের জন্য চলমান সময় একটি ধ্রুবক ফ্যাক্টর দ্বারা পৃথক হয় (অর্থাৎ, তাদের বৃদ্ধির হার একই), তাহলে কোড টিউনিংয়ের পার্থক্য চলমান সময়ের মধ্যে কোনো পার্থক্যের জন্য দায়ী হতে পারে। এই পরিস্থিতিতে অভিজ্ঞতামূলক তুলনার ব্যাপারে সন্দেহ পোষণ করুন।

বিশ্লেষণের আরেকটি পদ্ধতি হল সিমুলেশন। সিমুলেশনের ধারণা হল সমস্যাটিকে একটি কম্পিউটার প্রোগ্রামের সাথে মডেল করা এবং তারপরে ফলাফল পেতে এটি চালানো। অ্যালগরিদম বিশ্লেষণের প্রসঙ্গে, সিমুলেশন দুটি প্রতিযোগীর অভিজ্ঞতামূলক তুলনা থেকে আলাদা কারণ সিমুলেশনের উদ্দেশ্য হল বিশ্লেষণ করা যা অন্যথায় খুব কঠিন হতে পারে। এর একটি ভালো উদাহরণ চিত্র 9.8-এ প্রদর্শিত হয়েছে। এই চিত্রটি টেবিলে একটি বিনামূল্যের স্লট খুঁজতে ব্যবহৃত নীতির জন্য দুটি ভিন্ন অনুমানের অধীনে একটি হ্যাশ টেবিল থেকে একটি রেকর্ড সন্নিবেশ করা বা মুছে ফেলার খরচ দেখায়। y অক্ষ হল মূল্যায়ন করা হ্যাশ টেবিল স্লটের সংখ্যা, এবং x অক্ষ হল টেবিলে পূর্ণ স্লটের শতাংশ। এই বক্ররেখাগুলির জন্য গাণিতিক সমীকরণগুলি নির্ধারণ করা যেতে পারে, তবে এটি এত সহজ নয়। একটি যুক্তিসঙ্গত বিকল্প হল হ্যাশিং-এ সহজ বৈচিত্র্যগুলি লেখা। বিভিন্ন লোডিং অবস্থার জন্য প্রোগ্রামের খরচের সময় নির্ধারণ করে, চিত্র 9.8 এর মতো একটি প্লট তৈরি করা কঠিন নয়। এই বিশ্লেষণের উদ্দেশ্য হ্যাশিংয়ের কোন পদ্ধতি সবচেয়ে কার্যকর তা নির্ধারণ করা নয়, তাই আমরা হ্যাশিং বিকল্পগুলির অভিজ্ঞতামূলক তুলনা করছি না। পরিবর্তে, উদ্দেশ্য হল সঠিক লোডিং ফ্যাক্টর বিশ্লেষণ করা যা একটি দক্ষ হ্যাশিং সিস্টেমে সময় ব্যয় বনাম হ্যাশ টেবিলের আকারের (স্পেস খরচ) ভারসাম্য বজায় রাখতে ব্যবহার করা হবে।

3.12 আরও পড়া

অ্যালগরিদম বিশ্লেষণে অগ্রগামী কাজগুলির মধ্যে রয়েছে ডোনাল্ড ই. নুথের আর্ট অফ কম্পিউটার প্রোগ্রামিং [Knu97, Knu98], এবং Aho, Hopcroft, এবং Ullman [AHU74] এর কম্পিউটার অ্যালগরিদমের ডিজাইন এবং বিশ্লেষণ। Ω-এর বিকল্প সংজ্ঞা এসেছে [AHU83] থেকে। স্বরলিপির ব্যবহার "T(n) is in O(f(n))" এর পরিবর্তে সাধারণভাবে ব্যবহৃত "T(n) = O(f(n))" আমি ব্রাসার্ড এবং ব্র্যাটলি [BB96] থেকে উদ্ভূত, যদিও অবশ্যই এই ব্যবহার তাদের পূর্ববর্তী। অ্যালগরিদম বিশ্লেষণ কৌশল সম্পর্কে আরও তথ্যের জন্য পড়ার জন্য একটি ভাল বই কিসের সাথে তুলনা করা হয়? গ্রেগরি JE Rawlins [Raw92] দ্বারা।

বেন্টলে [বেন88] সংখ্যাগত বিশ্লেষণে একটি সমস্যা বর্ণনা করেছেন যার জন্য, 1945 এবং 1988-এর মধ্যে, সবচেয়ে পরিচিত অ্যালগরিদমের জটিলতা কমে গিয়েছিল
O(n) থেকে ^{7}) থেকে O(n ^{3}) n = 64 আকারের একটি সমস্যার জন্য, এটি মোটামুটি সমতুল্য
একই সময়ের মধ্যে কম্পিউটার হার্ডওয়্যারের সমস্ত অগ্রগতি থেকে অর্জিত গতি।

যদিও প্রোগ্রামের দক্ষতার সবচেয়ে গুরুত্বপূর্ণ দিক হল অ্যালগরিদম, একটি প্রোগ্রামের দক্ষ কোডিং থেকে অনেক উন্নতি করা যায়। দ্য মিথিক্যাল ম্যান-মন্থ [Bro95]-এ ফ্রেডার-ইক পি. ব্রুকস উদ্ধৃত করেছেন, একজন দক্ষ প্রোগ্রামার দশটি প্রোগ্রাম তৈরি করতে পারে যা একজন অদক্ষ প্রোগ্রামারের চেয়ে পাঁচগুণ দ্রুত চলে, এমনকি যখন কেউ তাদের কোডের গতি বাড়ানোর জন্য বিশেষ প্রচেষ্টা নেয় না। আপনার কোডিং দক্ষতা উন্নত করার জন্য চমৎকার এবং উপভোগযোগ্য প্রবন্ধের জন্য এবং আপনার কোডের গতি বাড়ানোর উপায়গুলির জন্য যখন এটি সত্যিই গুরুত্বপূর্ণ, জন বেন্টলি [Ben82, Ben00, Ben88] এর বইগুলি দেখুন। দ্য

উদাহরণ 3.18 এ বর্ণিত পরিস্থিতি দেখা দেয় যখন আমরা প্রকল্পে কাজ করছিলাম
[SU92] এ রিপোর্ট করা হয়েছে।

একটি আকর্ষণীয় হিসাবে, একটি সঠিক বাইনারি অনুসন্ধান অ্যালগরিদম লেখা সহজ নয়।
Knuth [Knu98] উল্লেখ করেছেন যে যখন প্রথম বাইনারি অনুসন্ধান 1946 সালে প্রকাশিত হয়েছিল, তখন
প্রথম বাগ-মুক্ত অ্যালগরিদম 1962 সাল পর্যন্ত প্রকাশিত হয়নি! বেন্টলি ("লেখা সঠিক
প্রোগ্রাম" [Ben00]) পাওয়া গেছে যে 90% কম্পিউটার পেশাদারদের তিনি পরীক্ষা করেছেন
দুই ঘন্টার মধ্যে একটি বাগ-মুক্ত বাইনারি অনুসন্ধান লিখতে পারেনি।

3.13 ব্যায়াম

3.1 চিত্র 3.1-এর পাঁচটি অভিব্যক্তির প্রতিটির জন্য, n-এর মানের পরিসর দিন
যার জন্য সেই অভিব্যক্তিটি সবচেয়ে কার্যকর।

3.2 নিম্নলিখিত অভিব্যক্তি গ্রাফ করুন। প্রতিটি অভিব্যক্তির জন্য, এর পরিসীমা উল্লেখ করুন
n এর মান যার জন্য সেই অভিব্যক্তিটি সবচেয়ে কার্যকর।

$4n^2$ $\log_3 n$ 3^n $20n$ $2^{\log_2 n}$ $n^{2/3}$

3.3 সবচেয়ে ধীর থেকে দ্রুততম বৃদ্ধির হার দ্বারা নিম্নলিখিত অভিব্যক্তিগুলি সাজান।

$4n^2$ $\log_3 n$ $n!$ 3^n $20n$ $2^{\log_2 n}$ $n^{2/3}$

n! শ্রেণীবিভাগে সহায়তার জন্য বিভাগ 2.2-এ স্টার্লিং-এর অনুমান দেখুন।

3.4 (a) ধরুন যে একটি নির্দিষ্ট অ্যালগরিদমের সময় জটিলতা রয়েছে $T(n) = 3 \times 2^n$, এবং এটি একটি নির্দিষ্ট মেশিনে এটির বাস্তবায়ন নির্বাহ করে
n ইনপুটের জন্য t সেকেন্ড সময় লাগে। এখন ধরুন আমরা একটি সঙ্গে উপস্থাপন করা হয়
মেশিন যা 64 গুণ দ্রুত। আমরা কত ইনপুট প্রক্রিয়া করতে পারে
টি সেকেন্ডের মধ্যে নতুন মেশিন?

(b) ধরুন যে অন্য একটি অ্যালগরিদমের সময় জটিলতা রয়েছে $T(n) = n^2$, এবং
যে একটি নির্দিষ্ট মেশিনে এটি একটি বাস্তবায়ন কার্যকর করা লাগে
n ইনপুটের জন্য t সেকেন্ড। এখন ধরুন যে আমাদের কাছে একটি মা-চাইন উপস্থাপন করা
হয়েছে যা 64 গুণ দ্রুত। আমরা কত ইনপুট প্রক্রিয়া করতে পারে
টি সেকেন্ডের মধ্যে নতুন মেশিন?

(c) একটি তৃতীয় অ্যালগরিদমের সময় জটিলতা $T(n) = 8n$। একটি নির্দিষ্ট মেশিনে এটির বাস্তবায়ন
কার্যকর করতে n ইনপুটগুলির জন্য t সেকেন্ড সময় লাগে।
একটি নতুন মেশিন দেওয়া হয়েছে যেটি 64 গুণ দ্রুত, কতগুলি ইনপুট দিতে পারে
আমরা টি সেকেন্ডে প্রক্রিয়া করি?

3.5 হার্ডওয়্যার বিক্রেতা XYZ কর্পোরেশন দাবি করেছে যে তাদের সর্বশেষ কম্পিউটার 100 চালাবে
তাদের প্রতিযোগী, Prunes, Inc এর চেয়ে গুণ বেশি দ্রুত। যদি Prunes, Inc.
কম্পিউটার এক ঘন্টার মধ্যে n সাইজের ইনপুটে একটি প্রোগ্রাম এক্সিকিউট করতে পারে, কি সাইজ

ইনপুট XYZ এর কম্পিউটার নিম্নলিখিত বৃদ্ধির হার সমীকরণ সহ প্রতিটি অ্যালগরিদমের জন্য এক ঘন্টার মধ্যে কার্যকর করতে পারে? n n2 n

$3 \quad 2^n$

3.6 (a) একটি বৃদ্ধির হার খুঁজুন যা রানের সময়কে বর্গ করে যখন আমরা ইনপুট আকার দ্বিগুণ করি। অর্থাৎ, যদি T(n) = X, তাহলে T(2n) = x (b) একটি বৃদ্ধির হার খুঁজুন যা ^{2}
রান টাইমকে কিউব করে যখন আমরা ইনপুট সাইজ দ্বিগুণ করি। অর্থাৎ, যদি T(n) = X, তাহলে T(2n) = x 3.7 big-Oh-এর সংজ্ঞা ব্যবহার করে দেখান যে 1 O(1) এ ^{3}
এবং 1 O(n) এ রয়েছে।

3.8 big-Oh এবং Ω-এর সংজ্ঞা ব্যবহার করে, নিম্নলিখিত রাশিগুলির জন্য উপরের এবং নীচের সীমাগুলি খুঁজুন। c এবং n0 এর জন্য উপযুক্ত মানগুলি উল্লেখ করতে ভুলবেন না ।

(a) c1n
(b) c2n ^{3} + c3
(c) c4n লগ n + c5n (d)
c62 ^{n} 6 + c7n

3.9 (a) k কি ক্ষুদ্রতম পূর্ণসংখ্যা যেমন n = O(n ^{k})?
(b) সবচেয়ে ছোট পূর্ণসংখ্যা k কি যেমন n log n = O(n 3.10 (a) কি 2n = ^{k})?
Θ(3n)? ব্যাখ্যা কর কেন বা কেন নয়। (b) কি 2 = Θ(3n)? ব্যাখ্যা কর
কেন বা কেন ^{n} না।

3.11 নিম্নলিখিত প্রতিটি জোড়া ফাংশনের জন্য, হয় f(n) হল O(g(n)), f(n) Ω(g(n) তে, অথবা f(n) = Θ(g(n)))। প্রতিটি জোড়ার জন্য, কোন সম্পর্ক-শিপ সঠিক তা নির্ধারণ করুন। অধ্যায় 3.4.5-এ আলোচিত সীমার পদ্ধতি ব্যবহার করে আপনার উত্তরকে ন্যায়সঙ্গত করুন।

(a) f(n) = লগ n ^{2}; g(n) = log n + 5. (b) f(n)
= n; g(n) = log n (c) f(n) = log2 n; ^{2}.
g(n) = লগ n. (d) f(n) = n; g(n) = log2n. (e)
f(n) = n লগ n + n; g(n) = লগ n. (f) f(n)
= লগ n (g) f(n) = 10; g(n) = লগ 10. (h) f(n) = 2n ;
g(n) = 10n (i) f(n) = ^{2}; g(n) = (লগ n) ^{2}.
2n ; g(n) = n লগ n. (j) f(n) = 2n ; g(n) =
3n । (k) f(n) = 2n ; g(n) = n ^{2}.

^{n} .

3.12 গড় ক্ষেত্রে নিম্নলিখিত কোড খণ্ডগুলির জন্য Θ নির্ধারণ করুন। অনুমান করুন যে সব ভেরিয়েবল টাইপ int হয়.

(a) a = b + c; d = a +
e;

(b) যোগফল = 0;
জন্য (i=0; i<3; i++)
(j=0; j<n; j++) যোগফল++;

(গ) যোগফল=0;
(i=0; i<n*n; i++) যোগফল++;

(d) জন্য (i=0; i < n-1; i++)
(j=i+1; j < n; j++) { tmp = A[i][j];

A[i][j] = A[j][i];
A[j][i] = tmp;
}

(e) যোগফল = 0;
জন্য (i=1; i<=n; i++)
(j=1; j<=n; j*=2) যোগফল++;

(f) যোগফল = 0;
জন্য (i=1; i<=n; i*=2)
(j=1; j<=n; j++) যোগফল++;

(g) অনুমান করুন যে অ্যারে A-তে n মান রয়েছে, র্যান্ডম ধ্রুবক সময় নেয় এবং সাজানোর জন্য n লগ n ধাপ লাগে।

(i=0; i<n; i++) {এর জন্য (j=0; j<n; j++)

A[i] = এলোমেলো(n); সাজান(A,
n);
}

(h) ধরুন অ্যারে A-তে 0 থেকে মানগুলির একটি এলোমেলো স্থানান্তর রয়েছে
n − 1.
যোগফল = 0;
(i=0; i<n; i++) এর জন্য (j=0; A[j]!
=i; j++) যোগফল++;

(i) যোগফল = 0; (i=0;
i<n; i++) যোগফল++ এর
জন্য যদি (EVEN(n)); অন্য

sum = যোগফল + n;

3.13 দেখান যে বিগ-থিটা নোটেশন (Θ) সেটে একটি সমতুল্য সম্পর্ককে সংজ্ঞায়িত করে
ফাংশন

3.14 নিচের কোড ফ্র্যাগমেন্টের জন্য সবচেয়ে ভালো নিম্ন সীমা দিন, যেমন a
n-এর প্রাথমিক মানের ফাংশন।

```
যখন (n > 1) যদি (ODD(n)) n
    = 3 * n + 1; else n = n/
        2;
```

আপনি কি মনে করেন যে উপরের সীমাটি নীচের সীমার জন্য আপনি যে উত্তর দিয়েছেন তার মতোই হতে পারে?

3.15 প্রতিটি অ্যালগরিদমে কি Θ চলমান সময়ের সমীকরণ থাকে? অন্য কথায়, চলমান সময়ের জন্য উপরের এবং নীচের সীমানা কি সবসময় একই থাকে?

3.16 যে সমস্ত সমস্যার জন্য কিছু অ্যালগরিদম বিদ্যমান তার কি একটি Θ চলমান সময়ের সমীকরণ আছে? অন্য কথায়, প্রতিটি সমস্যার জন্য, এবং ইনপুটগুলির যে কোনও নির্দিষ্ট শ্রেণীর জন্য, এমন কিছু অ্যালগরিদম আছে যার উপরের সীমাটি সমস্যার নিম্ন সীমার সমান?

3.17 মান দ্বারা ক্রমানুসারে একটি অ্যারে সংরক্ষণ পূর্ণসংখ্যা দেওয়া হয়েছে, যেখানে K অ্যারেতে একাধিকবার উপস্থিত হতে পারে এমন পরিস্থিতিতে K মান সহ প্রথম পূর্ণসংখ্যার অবস্থান ফেরাতে বাইনারি অনুসন্ধানের রুটিনটি পরিবর্তন করুন। নিশ্চিত হন যে আপনার অ্যালগরিদম হল Θ(log n), অর্থাৎ, K-এর ঘটনা পাওয়া গেলে অনুক্রমিক অনুসন্ধানের আশ্রয় নেবেন না।

3.18 মান অনুসারে পূর্ণসংখ্যা সঞ্চয় করার একটি অ্যারে দেওয়া, K-এর থেকে কম পূর্ণসংখ্যার অবস্থান ফেরাতে বাইনারি অনুসন্ধানের রুটিনটি পরিবর্তন করুন যখন K নিজেই অ্যারেতে উপস্থিত হয় না। যদি অ্যারের সর্বনিম্ন মান K-এর থেকে বেশি হয় তবে ERROR ফেরত দিন ।

3.19 অসীম আকারের একটি অ্যারেতে অনুসন্ধান সমর্থন করতে বাইনারি অনুসন্ধানের রুটিন পরিবর্তন করুন। বিশেষ করে, আপনাকে ইনপুট হিসাবে একটি সাজানো অ্যারে এবং অনুসন্ধান করার জন্য একটি কী মান K দেওয়া হয়েছে। অ্যারের সবচেয়ে ছোট মানের অবস্থান n কে কল করুন যা X এর সমান বা বড়। একটি অ্যালগরিদম প্রদান করুন যা সবচেয়ে খারাপ ক্ষেত্রে O(log n) তুলনাতে n নির্ধারণ করতে পারে। আপনার অ্যালগরিদম কেন প্রয়োজনীয় সময়সীমা পূরণ করে তা ব্যাখ্যা করুন।

3.20 বাইনারি অনুসন্ধানে আমরা যেভাবে বিভাজক বিন্দু বাছাই করি তা পরিবর্তন করা সম্ভব, এবং তারপরও একটি কার্যকরী অনুসন্ধানের রুটিন পেতে পারি। যাইহোক, যেখানে আমরা বিভাজন বিন্দু বাছাই করি অ্যালগরিদমের কর্মক্ষমতা প্রভাবিত করতে পারে।

(a) যদি আমরা ফাংশন বাইনারিতে বিভাজন বিন্দু গণনাকে i = (l + r)/2 থেকে i = (l + ((r − l)/3) তে পরিবর্তন করি), তাহলে সবচেয়ে খারাপ-কেস রানিং টাইম কী হবে? অ্যাসিম্পোটিক পদে হতে? যদি পার্থক্য শুধুমাত্র একটি ধ্রুবক সময়ের ফ্যাক্টর হয়, তবে পরিবর্তিত প্রোগ্রামটি বাইনারির মূল সংস্করণের সাথে কতটা ধীর বা দ্রুততর হবে ?

(b) যদি আমরা ফাংশন বাইনারিতে বিভাজক বিন্দু গণনাকে i = (l + r)/2 থেকে i = r − 2 তে পরিবর্তন করি, তাহলে অ্যাসিম্পটোটিক পদে সবচেয়ে খারাপ-কেস চলমান সময়টি কী হবে? যদি পার্থক্য শুধুমাত্র একটি ধ্রুবক সময়ের ফ্যাক্টর হয়, তবে পরিবর্তিত প্রোগ্রামটি বাইনারির মূল সংস্করণের সাথে কতটা ধীর বা দ্রুততর হবে ?

3.21 একটি জিগস পাজল একত্রিত করার জন্য একটি অ্যালগরিদম ডিজাইন করুন। অনুমান করুন যে প্রতিটি অংশের চারটি দিক রয়েছে এবং প্রতিটি অংশের চূড়ান্ত অভিযোজন পরিচিত (উপর, নীচে, ইত্যাদি)। ধরে নিন আপনার কাছে একটি ফাংশন আছে

bool compare(Piece a, Piece b, side ad) যা স্থির সময়ের মধ্যে বলতে পারে, একটি টুকরা a

এর পাশের বিজ্ঞাপনের অংশ b এর সাথে এবং b এর বিপরীত দিকের bd এর সাথে সংযোগ করে কিনা। আপনার অ্যালগরিদমের ইনপুটটিতে n এবং m মাত্রা সহ র‍্যান্ডম টুকরাগুলির একটি n × m অ্যারে থাকা উচিত। অ্যালগরিদমের টুকরাগুলিকে অ্যারেতে তাদের সঠিক অবস্থানে রাখা উচিত। আপনার অ্যালগরিদম অ্যাসিম্পোটিক অর্থে যতটা সম্ভব দক্ষ হওয়া উচিত। n টুকরাগুলিতে আপনার অ্যালগরিদমের চলমান সময়ের জন্য একটি সমষ্টি লিখুন এবং তারপর সমষ্টিটির জন্য একটি বন্ধ-ফর্ম সমাধান বের করুন।

3.22 একটি অ্যালগরিদমের গড় কেস খরচ কি সবচেয়ে খারাপ কেস খরচের চেয়ে খারাপ হতে পারে?
এটা কি সেরা কেস খরচের চেয়ে ভালো হতে পারে? কেন অথবা কেন নয় ব্যাখ্যা করুন।

3.23 প্রমাণ করুন যে একটি অ্যালগরিদম যদি গড় ক্ষেত্রে Θ(f(n)) হয়, তবে এটি Ω(f(n))
সবচেয়ে খারাপ অবস্থায়।

3.24 প্রমাণ করুন যে একটি অ্যালগরিদম যদি গড় ক্ষেত্রে Θ(f(n)) হয়, তবে এটি সর্বোত্তম ক্ষেত্রে O(f(n))।

3.14 প্রকল্প

3.1 কল্পনা করুন যে আপনি 32টি বুলিয়ান মান সঞ্চয় করার চেষ্টা করছেন এবং সেগুলিকে ঘন ঘন অ্যাক্সেস করতে হবে। একটি একক বিট ক্ষেত্র, একটি অক্ষর, একটি সংক্ষিপ্ত পূর্ণসংখ্যা বা একটি দীর্ঘ পূর্ণসংখ্যা হিসাবে বিকল্পভাবে সংরক্ষিত বুলিয়ান মানগুলি অ্যাক্সেস করার জন্য প্রয়োজনীয় সময়ের তুলনা করুন।
আপনার প্রোগ্রাম লেখার সময় দুটি বিষয় সতর্ক থাকতে হবে। প্রথমত, নিশ্চিত হোন যে আপনার প্রোগ্রামটি অর্থপূর্ণ পরিমাপ করার জন্য পর্যাপ্ত পরিবর্তনশীল অ্যাক্সেস করে। একটি একক অ্যাক্সেস চারটি পদ্ধতির পরিমাপের হারের চেয়ে অনেক ছোট। দ্বিতীয়ত, নিশ্চিত হোন যে আপনার প্রোগ্রামটি পরিবর্তনশীল অ্যাক্সেসে যতটা সম্ভব সময় ব্যয় করে অন্য জিনিস যেমন কলিং টাইমিং ফাংশন বা লুপ কাউন্টারগুলির জন্য ইনক্রিমেন্ট করার পরিবর্তে।

3.2 আপনার কম্পিউটারে অনুক্রমিক অনুসন্ধান এবং বাইনারি অনুসন্ধান অ্যালগরিদম প্রয়োগ করুন।
প্রতিটি অ্যালগরিদমের জন্য n = 10i আকারের অ্যারেতে টাইমিং চালান i এর জন্য 1 থেকে যতটা বড় একটি মান আপনার কম্পিউটারের মেমরি এবং কম্পাইলার অনুমতি দেবে। উভয় অ্যালগরিদমের জন্য, অ্যারেতে ক্রমানুসারে 0 থেকে n −1 মানগুলি সংরক্ষণ করুন এবং

প্রতিটি আকারে 0 থেকে n −1 পরিসরে বিভিন্ন র্যান্ডম অনুসন্ধান মান ব্যবহার করুন
n ফলাফলের সময়গুলি গ্রাফ করুন। যখন অনুক্রমিক অনুসন্ধান বাইনারি তুলনায় দ্রুত হয়
একটি সাজানো অ্যারের জন্য অনুসন্ধান?

3.3 ব্যায়াম 2.11-এ প্রদত্ত দুটি ফিবোনাচি সি-কোয়েন্স ফাংশনের জন্য রান করে এবং সময় দেয় এমন একটি প্রোগ্রাম বাস্তবায়ন করুন। ফলে চলমান গ্রাফ করুন
আপনার কম্পিউটার পরিচালনা করতে পারে হিসাবে n হিসাবে অনেক মান জন্য বার.