অ্যালগরিদম জটিলতা বিশ্লেষণের একটি মৃদু ভূমিকা

ডায়োনিসিস "ডায়নিজিজ" জিন্ড্রোস < dionyziz@gmail.com >

ভূমিকা

অনেক প্রোগ্রামার যারা আজকে কিছু চমৎকার এবং সবচেয়ে দরকারী সফ্টওয়্যার তৈরি করে, যেমন অনেকগুলি জিনিস যা আমরা ইন্টারনেটে দেখি বা প্রতিদিন ব্যবহার করি, তাদের একটি তাত্ত্বিক কম্পিউটার বিজ্ঞানের পটভূমি নেই। তারা এখনও বেশ দুর্দান্ত এবং সৃজনশীল প্রোগ্রামার এবং তারা যা তৈরি করে তার জন্য আমরা তাদের ধন্যবাদ জানাই।

যাইহোক, তাত্ত্বিক কম্পিউটার বিজ্ঞানের ব্যবহার এবং প্রয়োগ রয়েছে এবং এটি বেশ ব্যবহারিক হতে পারে। এই নিবন্ধে, প্রোগ্রামারদের লক্ষ্য করে যারা তাদের শিল্প জানেন কিন্তু যাদের কোনো তাত্ত্বিক কম্পিউটার বিজ্ঞানের পটভূমি নেই, আমি কম্পিউটার বিজ্ঞানের সবচেয়ে বাস্তবসম্মত টুলগুলির একটি উপস্থাপন করব: বিগ ও নোটেশন এবং অ্যালগরিদম জটিলতা বিশ্লেষণ। কম্পিউটার সায়েন্স একাডেমিক সেটিং এবং শিল্পে উত্পাদন-স্তরের সফ্টওয়্যার তৈরিতে উভয়ই কাজ করেছেন এমন একজন হিসাবে, এটি এমন একটি সরঞ্জাম যা আমি অনুশীলনে সত্যিকারের দরকারীগুলির মধ্যে একটি হিসাবে পেয়েছি, তাই আমি আশা করি এই নিবন্ধটি পড়ার পরে আপনি পারবেন এটি আরও ভাল করতে আপনার নিজের কোডে এটি প্রয়োগ করুন। এই পোস্টটি পড়ার পরে, আপনি কম্পিউটার বিজ্ঞানীদের "বিগ ও", "অসিম্পোটিক আচরণ" এবং "সবচেয়ে খারাপ-কেস বিশ্লেষণ" এর মতো সাধারণ শব্দগুলি বুঝতে সক্ষম হবেন।

এই পাঠ্যটি গ্রীস বা অন্য কোথাও আন্তর্জাতিক অলিম্পিয়াড ইনফরমেটিক্স , ছাত্রদের জন্য একটি অ্যালগরিদম প্রতিযোগিতা, বা অন্যান্য অনুরূপ প্রতিযোগিতায় আন্তর্জাতিকভাবে প্রতিদ্বন্দ্বিতাকারী জুনিয়র হাই স্কুল এবং উচ্চ বিদ্যালয়ের শিক্ষার্থীদের লক্ষ্য করে। যেমন, এটির কোনো গাণিতিক পূর্বশর্ত নেই এবং অ্যালগরিদমগুলির পিছনের তত্ত্বের দৃঢ় ধারণার সাথে অধ্যয়ন চালিয়ে যাওয়ার জন্য আপনাকে প্রয়োজনীয় পটভূমি দেবে। এই ছাত্রদের প্রতিযোগিতায় প্রতিদ্বন্দ্বিতা করতেন এমন একজন হিসাবে, আমি আপনাকে এই পুরো পরিচায়ক উপাদানটি পড়ার জন্য এবং এটি সম্পূর্ণরূপে বোঝার চেষ্টা করার জন্য আপনাকে পরামর্শ দিচ্ছি, কারণ আপনি অ্যালগরিদমগুলি অধ্যয়ন করতে এবং আরও উন্নত কৌশলগুলি শিখতে এটি প্রয়োজনীয় হবে।

আমি বিশ্বাস করি যে এই পাঠ্যটি শিল্প প্রোগ্রামারদের জন্য সহায়ক হবে যাদের তাত্ত্বিক কম্পিউটার বিজ্ঞানের সাথে খুব বেশি অভিজ্ঞতা নেই (এটি সত্য যে কিছু সবচেয়ে অনুপ্রেরণাদায়ক সফ্টওয়্যার ইঞ্জিনিয়ার কখনও কলেজে যাননি)। কিন্তু যেহেতু এটি ছাত্রদের জন্যও, তাই এটি মাঝে মাঝে পাঠ্যপুস্তকের মতো কিছুটা শোনাতে পারে। উপরন্তু, এই লেখার কিছু বিষয় আপনার কাছে খুব স্পষ্ট মনে হতে পারে; উদাহরণস্বরূপ, আপনি আপনার উচ্চ বিদ্যালয়ের বছরগুলিতে সেগুলি দেখে থাকতে পারেন। আপনি যদি মনে করেন যে আপনি সেগুলি বুঝতে পেরেছেন, আপনি সেগুলি এড়িয়ে যেতে পারেন। অন্যান্য বিভাগগুলি একটু বেশি গভীরতায় যায় এবং কিছুটা তাত্ত্বিক হয়ে ওঠে, কারণ এই প্রতিযোগিতায় অংশগ্রহণকারী শিক্ষার্থীদের গড় অনুশীলনকারীর চেয়ে তাত্ত্বিক অ্যালগরিদম সম্পর্কে আরও বেশি জানতে হবে। তবে এই জিনিসগুলি এখনও জানা ভাল এবং অনুসরণ করা খুব কঠিন নয়, তাই এটি সম্ভবত আপনার সময়ের জন্য উপযুক্ত। যেহেতু মূল পাঠ্যটি উচ্চ বিদ্যালয়ের শিক্ষার্থীদের লক্ষ্য করে তৈরি করা হয়েছিল, কোন গাণিতিক পটভূমির প্রয়োজন নেই, তাই প্রোগ্রামিং অভিজ্ঞতার সাথে যে কেউ (অর্থাৎ যদি আপনি জানেন যে পুনরাবৃত্তি কী) কোন সমস্যা ছাড়াই অনুসরণ করতে সক্ষম হবেন।

এই নিবন্ধটি জুড়ে, আপনি বিভিন্ন পয়েন্টার পাবেন যা আপনাকে আলোচনার বিষয়ের সুযোগের বাইরে আকর্ষণীয় উপাদানের সাথে সংযুক্ত করে। আপনি যদি একজন শিল্প প্রোগ্রামার হন তবে সম্ভবত আপনি এই ধারণাগুলির বেশিরভাগের সাথে পরিচিত। আপনি যদি প্রতিযোগিতায় অংশগ্রহণকারী একজন জুনিয়র ছাত্র হন, তাহলে এই লিঙ্কগুলি অনুসরণ করলে আপনি কম্পিউটার বিজ্ঞান বা সফ্টওয়্যার ইঞ্জিনিয়ারিংয়ের অন্যান্য ক্ষেত্রগুলি সম্পর্কে সূত্র পাবেন যেগুলি আপনি এখনও অন্বেষণ করেননি যা আপনি আপনার আগ্রহগুলিকে প্রসারিত করতে দেখতে পারেন।

বিগ ও স্বরলিপি এবং অ্যালগরিদম জটিলতা বিশ্লেষণ এমন একটি বিষয় যা অনেক শিল্প প্রোগ্রামার এবং জুনিয়র ছাত্রদের জন্য একইভাবে বোঝা কঠিন, ভয় করা বা সম্পূর্ণরূপে অকেজো হিসাবে এড়িয়ে যাওয়া। তবে এটি এতটা কঠিন বা তাত্ত্বিক নয় যতটা এটি প্রথমে মনে হতে পারে। অ্যালগরিদম জটিলতা আনুষ্ঠানিকভাবে একটি প্রোগ্রাম বা অ্যালগরিদম কত দ্রুত চলে তা পরিমাপ করার একটি উপায়, তাই এটি সত্যিই বেশ বাস্তবসম্মত। চলুন শুরু করা যাক টপিকটিকে একটু অনুপ্রাণিত করে।

প্রেরণা

আমরা ইতিমধ্যে জানি যে একটি প্রোগ্রাম কত দ্রুত চলে তা পরিমাপ করার জন্য সরঞ্জাম রয়েছে। *প্রোফাইলার* নামক প্রোগ্রাম রয়েছে যা মিলিসেকেন্ডে চলমান সময় পরিমাপ করে এবং বাধাগুলি চিহ্নিত করে আমাদের কোড অপ্টিমাইজ করতে সাহায্য করতে পারে। যদিও এটি একটি দরকারী টুল, এটি অ্যালগরিদম জটিলতার সাথে সত্যিই প্রাসঙ্গিক নয়। অ্যালগরিদম জটিলতা এমন কিছু যা ধারণা স্তরে দুটি অ্যালগরিদমের তুলনা করার জন্য ডিজাইন করা হয়েছে — নিম্ন-স্তরের বিবরণ যেমন বাস্তবায়ন প্রোগ্রামিং ভাষা, যে হার্ডওয়্যারটিতে অ্যালগরিদম চলে, বা প্রদত্ত CPU-এর নির্দেশনা সেট উপেক্ষা করে। আমরা অ্যালগরিদমগুলির তুলনা করতে চাই সেগুলি কী: কীভাবে কিছু গণনা করা হয় তার ধারণা। মিলিসেকেন্ড গণনা আমাদের এতে সাহায্য করবে না। এটা খুবই সম্ভব যে একটি নিম্ন-স্তরের প্রোগ্রামিং ভাষায় লেখা একটি খারাপ অ্যালগরিদম যেমন অ্যাসেম্বলি উচ্চ-স্তরের প্রোগ্রামিং ভাষা যেমন পাইথন বা রুবিতে লেখা একটি ভাল অ্যালগরিদমের চেয়ে অনেক দ্রুত চলে । সুতরাং এটি একটি "উন্নত অ্যালগরিদম" আসলে কি তা সংজ্ঞায়িত করার সময়।

যেহেতু অ্যালগরিদমগুলি এমন প্রোগ্রাম যা কেবলমাত্র একটি গণনা সম্পাদন করে, এবং কম্পিউটারগুলি প্রায়শই নেটওয়ার্কিং কাজগুলি বা ব্যবহারকারীর ইনপুট এবং আউটপুটের মতো অন্যান্য কাজ করে না, জটিলতা বিশ্লেষণ আমাদেরকে পরিমাপ করতে দেয় যে একটি প্রোগ্রাম কত দ্রুত গণনা সম্পাদন করে। বিশুদ্ধরূপে *গণনামূলক* ক্রিয়াকলাপের উদাহরণগুলির মধ্যে রয়েছে সংখ্যাসূচক ফ্লোটিং-পয়েন্ট অপারেশন যেমন যোগ এবং গুণন; একটি

প্রদত্ত মানের জন্য RAM এ ফিট করে এমন একটি ডাটাবেসের মধ্যে অনুসন্ধান করা; একটি ভিডিও গেমের মাধ্যমে একটি কৃত্রিম-বুদ্ধিমত্তা চরিত্রের পথ নির্ণয় করা যাতে তাদের ভার্চুয়াল জগতের মধ্যে অল্প দূরত্বে হাঁটতে হয় (**চিত্র 1** দেখুন); অথবা একটি স্ট্রিং এ একটি নিয়মিত এক্সপ্রেশন প্যাটার্ন ম্যাচ চলমান. স্পষ্টতই, কম্পিউটার প্রোগ্রামে গণনা সর্বব্যাপী।

জটিলতা বিশ্লেষণও একটি টুল যা আমাদের ব্যাখ্যা করতে দেয় যে ইনপুট বড় হওয়ার সাথে সাথে একটি অ্যালগরিদম কীভাবে আচরণ করে। যদি আমরা এটিকে একটি ভিন্ন ইনপুট খাওয়াই, তাহলে অ্যালগরিদম কীভাবে আচরণ করবে? যদি আমাদের অ্যালগরিদম 1000 আকারের একটি ইনপুট চালাতে 1 সেকেন্ড সময় নেয়, আমি ইনপুট আকার দ্বিগুণ করলে এটি কেমন আচরণ করবে? এটি কি ঠিক তত দ্রুত, অর্ধেক দ্রুত বা চার গুণ ধীর গতিতে চলবে? ব্যবহারিক প্রোগ্রামিং-এ, এটি গুরুত্বপূর্ণ কারণ এটি ইনপুট ডেটা বড় হলে আমাদের অ্যালগরিদম কীভাবে আচরণ করবে তা অনুমান করতে দেয়। উদাহরণস্বরূপ, যদি আমরা একটি ওয়েব অ্যাপ্লিকেশনের জন্য একটি অ্যালগরিদম তৈরি করি যা 1000 জন ব্যবহারকারীর সাথে ভালভাবে কাজ করে এবং এটির চলমান সময় পরিমাপ করে, অ্যালগরিদম জটিলতা বিশ্লেষণ ব্যবহার করে আমরা এর পরিবর্তে 2000 জন ব্যবহারকারী পেলে কী ঘটবে সে সম্পর্কে আমরা একটি সুন্দর ধারণা পেতে পারি। অ্যালগরিদমিক প্রতিযোগিতার জন্য, জটিলতা বিশ্লেষণ আমাদের প্রোগ্রামের সঠিকতা পরীক্ষা করতে ব্যবহৃত সবচেয়ে বড় টেস্টকেসগুলির জন্য আমাদের কোড কতক্ষণ চলবে সে সম্পর্কে আমাদের অন্তর্দৃষ্টি দেয়। তাই যদি আমরা একটি ছোট ইনপুটের জন্য আমাদের প্রোগ্রামের আচরণ পরিমাপ করে থাকি, তাহলে বড় ইনপুটের জন্য এটি কীভাবে আচরণ করবে সে সম্পর্কে আমরা একটি ভাল ধারণা পেতে পারি। একটি সাধারণ উদাহরণ দিয়ে শুরু করা যাক: একটি অ্যারের মধ্যে সর্বাধিক উপাদান খুঁজে বের করা।

নির্দেশাবলী গণনা

এই নিবন্ধে, আমি উদাহরণগুলির জন্য বিভিন্ন প্রোগ্রামিং ভাষা ব্যবহার করব। যাইহোক, আপনি যদি একটি নির্দিষ্ট প্রোগ্রামিং ভাষা না জানেন তবে হতাশ হবেন না। যেহেতু আপনি প্রোগ্রামিং জানেন, তাই আপনি পছন্দের প্রোগ্রামিং ভাষার সাথে পরিচিত না হলেও কোনো সমস্যা ছাড়াই উদাহরণগুলো পড়তে সক্ষম হবেন, কারণ সেগুলি সহজ হবে এবং আমি কোনো গোপন ভাষার বৈশিষ্ট্য ব্যবহার করব না। আপনি যদি অ্যালগরিদম প্রতিযোগিতায় প্রতিদ্বন্দ্বিতাকারী একজন ছাত্র হন, আপনি সম্ভবত C++ এর সাথে কাজ করেন , তাই আপনাকে অনুসরণ করতে কোনো সমস্যা হবে না। সেক্ষেত্রে আমি অনুশীলনের জন্য C++ ব্যবহার করে অনুশীলনে কাজ করার পরামর্শ দিই।

জাভাস্ক্রিপ্ট কোডের এই অংশের মতো একটি সাধারণ কোড ব্যবহার করে অ্যারের সর্বাধিক উপাদানটি সন্ধান করা যেতে পারে। *n* আকারের *একটি* ইনপুট অ্যারে দেওয়া হয়েছে :

```
1.var M = A[ 0 ];<font></font>

2.<font></font>

3.for ( var i = 0; i < n; ++i ) {<font></font>

4.if ( A[ i ] >= M ) {<font></font>

5.M = A[ i ];<font></font>

6.}<font></font>

7.}<font></font>
```

এখন, আমরা প্রথম জিনিসটি গণনা করব কতগুলি *মৌলিক নির্দেশাবলী* এই কোডটি কার্যকর করে। আমরা এটি শুধুমাত্র একবার করব এবং এটির প্রয়োজন হবে না যেহেতু আমরা আমাদের তত্ত্বটি বিকাশ করি, তাই আমরা যখন এটি করি তখন কয়েক মুহূর্তের জন্য আমার সাথে সহ্য করুন। আমরা কোডের এই অংশটি বিশ্লেষণ করার সময়, আমরা এটিকে সাধারণ নির্দেশাবলীতে বিভক্ত করতে চাই; যে জিনিসগুলি সরাসরি CPU দ্বারা নির্বাহ করা যেতে পারে - বা এর কাছাকাছি। আমরা অনুমান করব যে আমাদের প্রসেসর প্রতিটি একটি নির্দেশ হিসাবে নিম্নলিখিত ক্রিয়াকলাপগুলি সম্পাদন করতে পারে:

- একটি পরিবর্তনশীল একটি মান বরাদ্দ
- একটি অ্যারের মধ্যে একটি নির্দিষ্ট উপাদানের মান খুঁজছেন
- দুটি মান তুলনা
- একটি মান বৃদ্ধি
- মৌলিক গাণিতিক ক্রিয়াকলাপ যেমন যোগ এবং গুণ

আমরা অনুমান করব যে ব্রাঞ্চিং (কন্ডিশন মূল্যায়নের পরে কোডের মধ্যে পছন্দ `if`এবং অংশ) তাৎক্ষণিকভাবে ঘটে এবং এই নির্দেশাবলী গণনা করব না। উপরের কোডে, কোডের প্রথম লাইন হল:`elseif`

```
1.var M = A[ 0 ];
```

এর জন্য 2টি নির্দেশাবলীর প্রয়োজন: একটি A[0] খোঁজার জন্য এবং একটি *M* কে মান নির্ধারণ করার জন্য (আমরা ধরে নিচ্ছি যে n সর্বদা কমপক্ষে 1)। *n* এর মান নির্বিশেষে এই দুটি নির্দেশ সর্বদা অ্যালগরিদম দ্বারা প্রয়োজনীয় । লুপ `for`ইনিশিয়ালাইজেশন কোডও সবসময় চালাতে হয়। এটি আমাদের আরও দুটি নির্দেশ দেয়; একটি অ্যাসাইনমেন্ট এবং একটি তুলনা:

```
1.i = 0;<font></font>

2.i < n;<font></font>
```

এই প্রথম `for`লুপ পুনরাবৃত্তি আগে চালানো হবে. প্রতিটি লুপ পুনরাবৃত্তির পরে `for`, আমাদের চালানোর জন্য আরও দুটি নির্দেশাবলীর প্রয়োজন, *i-* এর একটি বৃদ্ধি এবং আমরা লুপে থাকব কিনা তা পরীক্ষা করার জন্য একটি তুলনা:

```
1.++i;<font></font>

2.i < n;<font></font>
```

সুতরাং, যদি আমরা লুপ বডি উপেক্ষা করি, তাহলে এই অ্যালগরিদমের প্রয়োজনীয় নির্দেশাবলীর সংখ্যা হল 4 + 2n। অর্থাৎ, লুপের শুরুতে 4টি নির্দেশাবলী `for`এবং প্রতিটি পুনরাবৃত্তির শেষে 2টি নির্দেশাবলী যার মধ্যে আমাদের *n* আছে । আমরা এখন একটি গাণিতিক ফাংশন f(n) সংজ্ঞায়িত করতে পারি যেটি, একটি *n* দিলে , আমাদের অ্যালগরিদমের প্রয়োজনীয় নির্দেশাবলীর সংখ্যা দেয়। একটি খালি `for`শরীরের জন্য, আমাদের আছে f(n) = 4 + 2n।

সবচেয়ে খারাপ ক্ষেত্রে বিশ্লেষণ

এখন, `for`শরীরের দিকে তাকিয়ে, আমাদের কাছে একটি অ্যারে লুকআপ অপারেশন রয়েছে এবং একটি তুলনা যা সর্বদা ঘটে:

```
1.if ( A[ i ] >= M ) { ...
```

যে ডান সেখানে দুটি নির্দেশ. কিন্তু `if`অ্যারের মান আসলে কি তার উপর নির্ভর করে বডি চলতে পারে বা নাও চলতে পারে। যদি এটি এমন হয় `A[ i ] >= M`, তাহলে আমরা এই দুটি অতিরিক্ত নির্দেশনা চালাব — একটি অ্যারে লুকআপ এবং একটি অ্যাসাইনমেন্ট:

```
1.M = A[ i ]
```

কিন্তু এখন আমরা একটি f(n) কে সহজে সংজ্ঞায়িত করতে পারি না, কারণ আমাদের নির্দেশের সংখ্যা শুধুমাত্র n এর উপর নির্ভর করে না কিন্তু আমাদের ইনপুটের উপরও নির্ভর করে। উদাহরণস্বরূপ, `A = [ 1, 2, 3, 4 ]`অ্যালগরিদমের জন্য এর চেয়ে বেশি নির্দেশাবলীর প্রয়োজন হবে `A = [ 4, 3, 2, 1 ]`। অ্যালগরিদম বিশ্লেষণ করার সময়, আমরা প্রায়শই সবচেয়ে খারাপ পরিস্থিতি বিবেচনা করি। আমাদের অ্যালগরিদমের জন্য সবচেয়ে খারাপ কি ঘটতে পারে? কখন আমাদের অ্যালগরিদম সম্পূর্ণ করার জন্য সর্বাধিক নির্দেশাবলীর প্রয়োজন হয়? এই ক্ষেত্রে, যখন আমাদের ক্রমবর্ধমান ক্রমে একটি অ্যারে থাকে যেমন `A = [ 1, 2, 3, 4 ]`. সেক্ষেত্রে, *M কে* প্রতিবার প্রতিস্থাপন করতে হবে এবং যাতে সর্বাধিক নির্দেশনা পাওয়া যায়। কম্পিউটার বিজ্ঞানীদের কাছে এর জন্য একটি অভিনব নাম রয়েছে এবং তারা এটিকে *সবচেয়ে খারাপ-কেস বিশ্লেষণ* বলে ; আমরা সবচেয়ে দুর্ভাগা যখন শুধুমাত্র কেস বিবেচনা করা ছাড়া আর কিছুই নয়। সুতরাং, সবচেয়ে খারাপ ক্ষেত্রে, আমাদের `for`শরীরের মধ্যে চালানোর জন্য 4টি নির্দেশাবলী রয়েছে, তাই আমাদের আছে f(n) = 4 + 2n + 4n = 6n + 4। এই ফাংশন f, একটি সমস্যা আকার n দেওয়া হয়েছে, আমাদের সংখ্যা দেয় নির্দেশাবলী যা সবচেয়ে খারাপ ক্ষেত্রে প্রয়োজন হবে।

অ্যাসিম্পোটিক আচরণ

এই ধরনের একটি ফাংশন দেওয়া, আমরা একটি চমত্কার ভাল ধারণা আছে একটি অ্যালগরিদম কত দ্রুত. যাইহোক, যেমন আমি প্রতিশ্রুতি দিয়েছিলাম, আমাদের প্রোগ্রামে নির্দেশাবলী গণনা করার ক্লান্তিকর কাজের মধ্যে দিয়ে যেতে হবে না। এছাড়াও, প্রতিটি প্রোগ্রামিং ভাষার স্টেটমেন্টের জন্য প্রয়োজনীয় প্রকৃত CPU নির্দেশাবলীর সংখ্যা নির্ভর করে আমাদের প্রোগ্রামিং ভাষার কম্পাইলার এবং উপলব্ধ CPU নির্দেশনা সেটের উপর (যেমন এটি আপনার পিসিতে একটি AMD বা একটি ইন্টেল পেন্টিয়াম, বা আপনার প্লেস্টেশনে একটি MIPS প্রসেসর। 2) এবং আমরা বলেছিলাম যে আমরা এটি উপেক্ষা করব। আমরা এখন একটি "ফিল্টার" এর মাধ্যমে আমাদের "f" ফাংশন চালাব যা আমাদের সেই ছোটখাটো বিবরণগুলি থেকে পরিত্রাণ পেতে সাহায্য করবে যা কম্পিউটার বিজ্ঞানীরা উপেক্ষা করতে পছন্দ করেন।

আমাদের ফাংশনে, 6n + 4, আমাদের দুটি পদ আছে: 6n এবং 4। জটিলতা বিশ্লেষণে আমরা কেবলমাত্র প্রোগ্রাম ইনপুট (*n*) বড় হওয়ার সাথে সাথে নির্দেশ-গণনা ফাংশনের কী হবে তা নিয়ে চিন্তা করি। এটি সত্যিই "সবচেয়ে খারাপ পরিস্থিতি" আচরণের পূর্ববর্তী ধারণাগুলির সাথে যায়: আমরা খারাপ আচরণ করলে আমাদের অ্যালগরিদম কীভাবে আচরণ করে তা নিয়ে আগ্রহী; যখন কঠিন কিছু করতে চ্যালেঞ্জ করা হয়। লক্ষ্য করুন যে অ্যালগরিদম তুলনা করার সময় এটি সত্যিই দরকারী। যদি একটি অ্যালগরিদম একটি বড় ইনপুটের জন্য অন্য অ্যালগরিদমকে হারায়, তবে এটি সম্ভবত সত্য যে একটি সহজ, ছোট ইনপুট দেওয়া হলে দ্রুত অ্যালগরিদমটি দ্রুত থাকে। **আমরা যে পদগুলি বিবেচনা করছি, আমরা সেই সমস্ত পদগুলিকে বাদ দেব যেগুলি ধীরে ধীরে বৃদ্ধি পায় এবং শুধুমাত্র n বড় হওয়ার সাথে সাথে দ্রুত বাড়তে থাকা শর্তগুলিকে রাখব।** *n* বড় হওয়ার সাথে সাথে স্পষ্টতই 4 একটি 4 থেকে যায় , কিন্তু 6n বড় থেকে বড় হয়, তাই এটি বৃহত্তর সমস্যার জন্য আরও বেশি গুরুত্ব দেয়। অতএব, আমরা প্রথমে 4টি ড্রপ করব এবং ফাংশনটিকে f(n) = 6n হিসাবে রাখব।

আপনি যদি এটি সম্পর্কে চিন্তা করেন তবে এটি বোধগম্য হয়, কারণ 4 কেবল একটি "সূচনা ধ্রুবক"। বিভিন্ন প্রোগ্রামিং ল্যাঙ্গুয়েজ সেট আপ করতে আলাদা সময় লাগতে পারে। উদাহরণস্বরূপ, জাভা এর ভার্চুয়াল মেশিন শুরু করতে কিছু সময় প্রয়োজন । যেহেতু আমরা প্রোগ্রামিং ভাষার পার্থক্য উপেক্ষা করছি, তাই এই মানটিকে উপেক্ষা করাই বোধগম্য।

দ্বিতীয় যে জিনিসটি আমরা উপেক্ষা করব তা হল n এর সামনে ধ্রুবক গুণক , এবং তাই আমাদের ফাংশন f(n) = n হয়ে যাবে। আপনি দেখতে পাচ্ছেন যে এটি জিনিসগুলিকে অনেক সহজ করে তোলে। আবার, এই গুনগত ধ্রুবকটি বাদ দেওয়া কিছুটা বোধগম্য হয় যদি আমরা চিন্তা করি কিভাবে বিভিন্ন প্রোগ্রামিং ভাষা সংকলন করে। একটি ভাষায় "অ্যারে লুকআপ" বিবৃতি বিভিন্ন প্রোগ্রামিং ভাষায় বিভিন্ন নির্দেশাবলীতে কম্পাইল হতে পারে। উদাহরণ স্বরূপ, C-তে করা `A[ i ]`একটি চেক অন্তর্ভুক্ত করে না যে *i* ঘোষিত অ্যারের আকারের মধ্যে আছে, যখন Pascal- এ এটি আছে। সুতরাং, নিম্নলিখিত প্যাসকেল কোড:

```
1.M := A[ i ]
```

C-তে নিম্নলিখিতগুলির সমতুল্য:

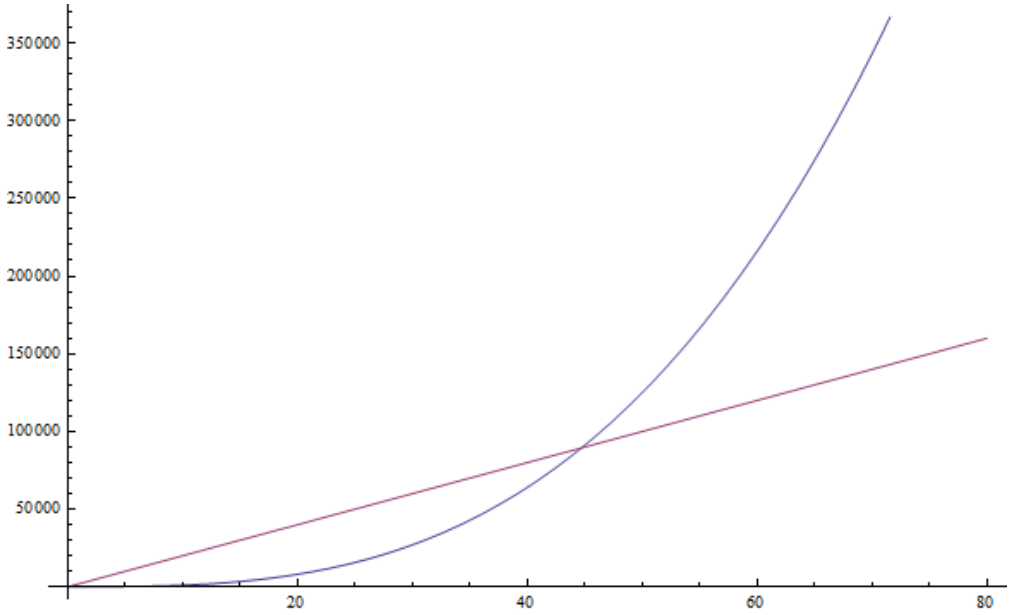
```
1.if ( i >= 0 && i < n ) {<font></font>

2.M = A[ i ];<font></font>

3.}<font></font>
```

সুতরাং এটা আশা করা যুক্তিসঙ্গত যে বিভিন্ন প্রোগ্রামিং ল্যাঙ্গুয়েজ যখন আমরা তাদের নির্দেশাবলী গণনা করি তখন বিভিন্ন ফ্যাক্টর তৈরি করবে। আমাদের উদাহরণে যেখানে আমরা প্যাসকেলের জন্য একটি বোবা কম্পাইলার ব্যবহার করছি যেটি সম্ভাব্য অপ্টিমাইজেশান সম্পর্কে অজ্ঞ, Pascal-এর প্রতিটি অ্যারে অ্যাক্সেসের জন্য 1টি নির্দেশনা C এর পরিবর্তে 3টি নির্দেশাবলী প্রয়োজন৷ এই ফ্যাক্টর বাদ দেওয়া নির্দিষ্ট প্রোগ্রামিং ভাষা এবং কম্পাইলার মধ্যে পার্থক্য উপেক্ষা এবং শুধুমাত্র অ্যালগরিদম নিজেই ধারণা বিশ্লেষণ লাইন বরাবর যায়.

উপরে বর্ণিত হিসাবে "সমস্ত ফ্যাক্টর বাদ দেওয়া" এবং "বৃহত্তর ক্রমবর্ধমান শব্দ বজায় রাখার" এই ফিল্টারটিকে আমরা *অ্যাসিম্পোটিক আচরণ* বলি । সুতরাং f(n) = 2n + 8 এর অ্যাসিম্পটোটিক আচরণ f(n) = n ফাংশন দ্বারা বর্ণিত হয়েছে। গাণিতিকভাবে বলতে গেলে, আমরা এখানে যা বলছি তা হল যে আমরা ফাংশনের সীমাতে আগ্রহী কারণ *n* অনন্তের দিকে ঝোঁক; কিন্তু যদি আপনি বুঝতে না পারেন যে এই বাক্যাংশটির আনুষ্ঠানিক অর্থ কী, চিন্তা করবেন না, কারণ এটিই আপনার জানা দরকার। (একটি পাশের নোটে, একটি কঠোর গাণিতিক সেটিংয়ে, আমরা সীমার মধ্যে ধ্রুবকগুলি বাদ দিতে সক্ষম হব না; কিন্তু কম্পিউটার বিজ্ঞানের উদ্দেশ্যে, আমরা উপরে বর্ণিত কারণগুলির জন্য এটি করতে চাই।) আসুন কয়েকটি উদাহরণের কাজ করি ধারণার সাথে নিজেদের পরিচিত করুন।



ধ্রুবক গুণনীয়কগুলিকে বাদ দিয়ে এবং সবচেয়ে দ্রুত বাড়তে থাকা পদগুলিকে রেখে নিচের উদাহরণ ফাংশনের অ্যাসিম্পোটিক আচরণ খুঁজে বের করা যাক।

1. f(n) = 5n + 12 দেয় f(n) = n।

   উপরের মত ঠিক একই যুক্তি ব্যবহার করে.

2. f(n) = 109 দেয় f(n) = 1।

   আমরা গুণক 109 * 1 বাদ দিচ্ছি, কিন্তু এই ফাংশনের একটি অ-শূন্য মান রয়েছে তা নির্দেশ করার জন্য আমাদের এখানে 1 রাখতে হবে।

3. f(n) = n^2 + 3n + 112 দেয় f(n) = n^2

   এখানে, n^2 যথেষ্ট বড় n এর জন্য 3n এর থেকে বড় হয়, তাই আমরা এটি রাখছি।

4. f(n) = n^3 + 1999 n + 1337 দেয় f(n) = n^3

   যদিও n-এর সামনের ফ্যাক্টরটি বেশ বড়, তবুও আমরা যথেষ্ট বড় *n* খুঁজে পেতে পারি যাতে n^3 1999n এর থেকে বড় হয়। যেহেতু আমরা *n* এর খুব বড় মানের জন্য আচরণে আগ্রহী , আমরা শুধুমাত্র n^3 রাখি (**চিত্র 2** দেখুন)।

5. f(n) = n + $\sqrt{n}$ দেয় f(n) = n

   এটি তাই কারণ n $\sqrt{n}$ যত দ্রুত বৃদ্ধি পায় তার *চেয়ে* দ্রুত বৃদ্ধি পায় ।

আপনি নিজেরাই নিম্নলিখিত উদাহরণগুলি চেষ্টা করতে পারেন:

অনুশীলনী 1

1. $f(n) = n^6 + 3n$
2. $f(n) = 2^n + 12$
3. $f(n) = 3^n + 2^n$
4. $f(n) = n^n + n$

(আপনার ফলাফল লিখুন; সমাধান নীচে দেওয়া আছে)

আপনি যদি উপরের একটির সাথে সমস্যায় পড়েন তবে কিছু বড় *n* প্লাগ ইন করুন এবং দেখুন কোন শব্দটি বড়। বেশ সোজা, হাহ?

জটিলতা

সুতরাং এটি আমাদেরকে যা বলছে তা হল যেহেতু আমরা এই সমস্ত আলংকারিক ধ্রুবকগুলিকে বাদ দিতে পারি, এটি একটি প্রোগ্রামের নির্দেশ-গণনা ফাংশনের অ্যাসিম্পোটিক আচরণ বলা বেশ সহজ। প্রকৃতপক্ষে, যে কোনো প্রোগ্রামে কোনো লুপ নেই $f(n) = 1$ থাকবে, যেহেতু এটির প্রয়োজনীয় নির্দেশাবলীর সংখ্যা শুধুমাত্র একটি ধ্রুবক (যদি না এটি পুনরাবৃত্তি ব্যবহার করে; নীচে দেখুন)। একটি একক লুপ সহ যেকোনো প্রোগ্রাম যা 1 থেকে *n* পর্যন্ত যায় তার $f(n) = n$ থাকবে, যেহেতু এটি লুপের আগে একটি ধ্রুবক সংখ্যক নির্দেশনা, লুপের পরে একটি ধ্রুবক সংখ্যক নির্দেশাবলী এবং এর মধ্যে একটি ধ্রুবক সংখ্যক নির্দেশনা করবে। লুপ যা সব *n* বার চালানো.

এটি এখন স্বতন্ত্র নির্দেশাবলী গণনা করার চেয়ে অনেক সহজ এবং কম ক্লান্তিকর হওয়া উচিত, তাই এর সাথে পরিচিত হওয়ার জন্য কয়েকটি উদাহরণ দেখে নেওয়া যাক। নিম্নলিখিত PHP প্রোগ্রামটি *n* আকারের একটি অ্যারের মধ্যে একটি নির্দিষ্ট মান বিদ্যমান কিনা তা পরীক্ষা করে *:*

```
01.<?php<font></font>

02.$exists = false;<font></font>

03.for ( $i = 0; $i < n; ++$i ) {<font></font>

04.if ( $A[ $i ] == $value ) {<font></font>

05.$exists = true;<font></font>

06.break;<font></font>

07.}<font></font>

08.}<font></font>

09.?><font></font>
```

একটি অ্যারের মধ্যে একটি মান অনুসন্ধানের এই পদ্ধতিটিকে *রৈখিক অনুসন্ধান* বলা হয় । এটি একটি যুক্তিসঙ্গত নাম, কারণ এই প্রোগ্রামটিতে রয়েছে $f(n) = n$ (আমরা পরবর্তী বিভাগে "রৈখিক" এর অর্থ ঠিক কী তা সংজ্ঞায়িত করব)। আপনি লক্ষ্য করতে পারেন যে এখানে একটি "ব্রেক" বিবৃতি রয়েছে যা একটি একক পুনরাবৃত্তির পরেও

প্রোগ্রামটিকে শীঘ্রই বন্ধ করে দিতে পারে। কিন্তু প্রত্যাহার করুন যে আমরা সবচেয়ে খারাপ-কেস পরিস্থিতিতে আগ্রহী, যা এই প্রোগ্রামের জন্য *অ্যারের* জন্য মান ধারণ না করার জন্য। তাই আমাদের এখনও f(n) = n আছে।

ব্যায়াম 2

উপরোক্ত PHP প্রোগ্রামের জন্য প্রয়োজনীয় নির্দেশাবলীর সংখ্যা পদ্ধতিগতভাবে বিশ্লেষণ করুন f(n) খুঁজে বের করার জন্য *n*- এর ক্ষেত্রে, একইভাবে আমরা কীভাবে আমাদের প্রথম জাভাস্ক্রিপ্ট প্রোগ্রাম বিশ্লেষণ করেছি। তারপর যাচাই করুন যে, লক্ষণীয়ভাবে, আমাদের কাছে f(n) = n আছে।

আসুন একটি পাইথন প্রোগ্রাম দেখি যা একটি যোগফল তৈরি করতে দুটি অ্যারে উপাদান যোগ করে যা এটি অন্য ভেরিয়েবলে সংরক্ষণ করে:

```
1.v = a[ 0 ] + a[ 1 ]
```

এখানে আমাদের একটি ধ্রুবক সংখ্যক নির্দেশ রয়েছে, তাই আমাদের আছে f(n) = 1।

C++ এ নিম্নলিখিত প্রোগ্রামটি *n* আকারের *A* নামের একটি ভেক্টর (একটি অভিনব অ্যারে) এর মধ্যে কোথাও একই দুটি মান রয়েছে কিনা তা পরীক্ষা করে:

```
01.bool duplicate = false;<font></font>

02.for ( int i = 0; i < n; ++i ) {<font></font>

03.for ( int j = 0; j < n; ++j ) {<font></font>

04.if ( i != j && A[ i ] == A[ j ] ) {<font></font>

05.duplicate = true;<font></font>

06.break;<font></font>

07.}<font></font>

08.}<font></font>

09.if ( duplicate ) {<font></font>

10.break;<font></font>

11.}<font></font>

12.}<font></font>
```

যেহেতু এখানে আমাদের একে অপরের মধ্যে দুটি নেস্টেড লুপ রয়েছে, আমাদের f(n) = n 2 দ্বারা বর্ণিত একটি অ্যাসিম্পোটিক আচরণ থাকবে ।

অঙ্গুষ্ঠের নিয়ম : প্রোগ্রামের নেস্টেড লুপগুলি গণনা করে সহজ প্রোগ্রামগুলি বিশ্লেষণ করা যেতে পারে। n আইটেমের উপর একটি একক লুপ f(n) = n দেয়। একটি লুপের মধ্যে একটি লুপ f(n) = n^2 দেয় । একটি লুপের মধ্যে একটি লুপের মধ্যে একটি লুপ f(n) = n^3 দেয় ।

যদি আমাদের একটি প্রোগ্রাম থাকে যা একটি লুপের মধ্যে একটি ফাংশনকে কল করে এবং আমরা জানি যে ফাংশনটি কতগুলি নির্দেশাবলী সম্পাদন করে, তাহলে পুরো প্রোগ্রামের নির্দেশাবলীর সংখ্যা নির্ধারণ করা সহজ। প্রকৃতপক্ষে, আসুন এই C উদাহরণটি একবার দেখে নেওয়া যাক:

```
1.int i;<font></font>

2.for ( i = 0; i < n; ++i ) {<font></font>

3.f( n );<font></font>

4.}<font></font>
```

যদি আমরা জানি যে এটি এমন `f( n )`একটি ফাংশন যা ঠিক *n* নির্দেশাবলী সম্পাদন করে, তাহলে আমরা জানতে পারি যে পুরো প্রোগ্রামের নির্দেশাবলীর সংখ্যাটি লক্ষণীয়ভাবে n^2 , কারণ ফাংশনটিকে ঠিক *n* বার বলা হয়।

অঙ্গুষ্ঠের নিয়ম : অনুক্রমিক লুপগুলির একটি সিরিজ দেওয়া হলে, তাদের মধ্যে সবচেয়ে ধীরটি প্রোগ্রামের অ্যাসিম্পোটিক আচরণ নির্ধারণ করে। একটি একক লুপের পরে দুটি নেস্টেড লুপগুলি অ্যাসিম্পটোটিকভাবে একা নেস্টেড লুপের মতোই, কারণ নেস্টেড লুপগুলি সাধারণ লুপের *উপর আধিপত্য করে ।*

এখন, কম্পিউটার বিজ্ঞানীরা যে অভিনব স্বরলিপি ব্যবহার করেন তার দিকে সুইচ করা যাক। যখন আমরা নির্ভুলভাবে ঠিক যেমন f নির্ণয় করেছি, তখন আমরা বলব যে আমাদের প্রোগ্রাম হল Θ(f(n))। উদাহরণস্বরূপ, উপরের প্রোগ্রামগুলি যথাক্রমে Θ(1), Θ(n^2) এবং Θ(n^2)। Θ(n) উচ্চারিত হয় "theta of n"। কখনও কখনও আমরা বলি যে f(n), ধ্রুবক সহ নির্দেশগুলি গণনা করার মূল ফাংশনটি হল Θ(কিছু)। উদাহরণস্বরূপ, আমরা বলতে পারি যে f(n) = 2n একটি ফাংশন যা Θ(n) — এখানে নতুন কিছু নেই। আমরা 2n ∈ Θ(n)ও লিখতে পারি, যার উচ্চারণ হয় "two n is theta of n"। এই স্বরলিপি সম্পর্কে বিভ্রান্ত হবেন না: শুধু এটাই বলছে যে আমরা যদি একটি প্রোগ্রামের জন্য প্রয়োজনীয় নির্দেশাবলীর সংখ্যা গণনা করি এবং সেগুলি 2n হয়, তাহলে আমাদের অ্যালগরিদমের অ্যাসিম্পটোটিক আচরণ n দ্বারা বর্ণনা করা হয়, যা আমরা ধ্রুবকগুলি বাদ দিয়ে পেয়েছি . এই স্বরলিপি দেওয়া, নিম্নলিখিত কিছু সত্য গাণিতিক বিবৃতি আছে:

1. $n^6 + 3n \in \Theta(n^6)$
2. $2^n + 12 \in \Theta(2^n)$
3. $3^n + 2^n \in \Theta(3^n)$
4. $n^n + n \in \Theta(n^n)$

যাইহোক, আপনি যদি উপরে থেকে ব্যায়াম 1 সমাধান করেন, তাহলে এই উত্তরগুলিই আপনার পাওয়া উচিত ছিল।

আমরা এই ফাংশনটিকে বলি, অর্থাৎ যাকে আমরা Θ(এখানে), *সময় জটিলতা* বা আমাদের অ্যালগরিদমের *জটিলতা* বলি। সুতরাং Θ(n) সহ একটি অ্যালগরিদম হল জটিলতা n। আমাদের কাছে Θ(1), Θ(n), Θ(n^2) এবং Θ(log(n)) এর জন্যও বিশেষ নাম রয়েছে কারণ এগুলি প্রায়শই ঘটে। আমরা বলি যে একটি Θ(1) অ্যালগরিদম

একটি *ধ্রুব-সময় অ্যালগরিদম* , Θ(n) *রৈখিক* , Θ(n^2) *দ্বিঘাত* এবং Θ(log(n)) *লগারিদমিক* (চিন্তা করবেন না যদি আপনি না করেন লগারিদমগুলি এখনও কী তা জানুন - আমরা এক মিনিটের মধ্যে এটিতে পৌঁছাব)।

অঙ্গুষ্ঠের নিয়ম : একটি বড় Θ সহ প্রোগ্রামগুলি ছোট Θ সহ প্রোগ্রামগুলির চেয়ে ধীর গতিতে চলে।

বিগ-ও স্বরলিপি

এখন, এটি কখনও কখনও সত্য যে এই ফ্যাশনে একটি অ্যালগরিদমের আচরণ সঠিকভাবে বের করা কঠিন হবে যেমন আমরা উপরে করেছি, বিশেষ করে আরও জটিল উদাহরণের জন্য। যাইহোক, আমরা বলতে সক্ষম হব যে আমাদের অ্যালগরিদমের আচরণ কখনই একটি নির্দিষ্ট সীমা অতিক্রম করবে না। এটি আমাদের জন্য জীবনকে সহজ করে তুলবে, কারণ আমাদের অ্যালগরিদম ঠিক কতটা দ্রুত চলে তা নির্দিষ্ট করতে হবে না, এমনকি ধ্রুবককে উপেক্ষা করার সময়ও আমরা আগের মতো করেছিলাম। আমাদের যা করতে হবে তা হল একটি নির্দিষ্ট সীমা খুঁজে। এটি একটি উদাহরণ দিয়ে সহজে ব্যাখ্যা করা হয়েছে।

একটি বিখ্যাত সমস্যা কম্পিউটার বিজ্ঞানীরা অ্যালগরিদম শেখানোর জন্য ব্যবহার করেন *বাছাই সমস্যা* । সাজানোর সমস্যায়, *n* আকারের *একটি* অ্যারে দেওয়া হয় (পরিচিত শোনায়?) এবং আমাদের এই অ্যারে সাজানোর একটি প্রোগ্রাম লিখতে বলা হয়। এই সমস্যাটি আকর্ষণীয় কারণ এটি বাস্তব সিস্টেমে একটি বাস্তব সমস্যা। উদাহরণস্বরূপ, একটি ফাইল এক্সপ্লোরারকে এটি প্রদর্শিত ফাইলগুলিকে নাম অনুসারে সাজাতে হবে যাতে ব্যবহারকারী সহজেই সেগুলি নেভিগেট করতে পারে। অথবা, অন্য একটি উদাহরণ হিসাবে, একটি ভিডিও গেমের জন্য ভার্চুয়াল জগতের মধ্যে খেলোয়াড়ের চোখ থেকে তাদের দূরত্বের উপর ভিত্তি করে বিশ্বে প্রদর্শিত 3D বস্তুগুলিকে সাজানোর প্রয়োজন হতে পারে যা দৃশ্যমান এবং কী নয় তা নির্ধারণ করতে, যাকে দৃশ্যমান সমস্যা বলা হয় (**চিত্র 3** দেখুন)। যে বস্তুগুলি প্লেয়ারের সবচেয়ে কাছাকাছি হতে দেখা যায় সেগুলিই দৃশ্যমান, যখন যেগুলি আরও আছে সেগুলি তাদের সামনে থাকা বস্তুগুলি দ্বারা লুকিয়ে যেতে পারে। বাছাই করাও আকর্ষণীয় কারণ এটি সমাধান করার জন্য অনেক অ্যালগরিদম রয়েছে, যার মধ্যে কিছু অন্যদের চেয়ে খারাপ। এটি সংজ্ঞায়িত করা এবং ব্যাখ্যা করার জন্য একটি সহজ সমস্যা। সুতরাং আসুন কোডের একটি অংশ লিখি যা একটি অ্যারে সাজায়।

এখানে রুবিতে একটি অ্যারে সাজানো বাস্তবায়নের একটি অদক্ষ উপায়। (অবশ্যই, রুবি বিল্ড-ইন ফাংশন ব্যবহার করে অ্যারে সাজানো সমর্থন করে যা আপনার পরিবর্তে ব্যবহার করা উচিত, এবং যা আমরা এখানে যা দেখব তার

থেকে অবশ্যই দ্রুত। তবে এটি এখানে চিত্রিত করার উদ্দেশ্যে।)

```
01.b = []<font></font>

02.n.times do<font></font>

03.m = a[ 0 ]<font></font>

04.mi = 0<font></font>

05.a.each_with_index do |element, i|<font></font>

06.if element < m<font></font>

07.m = element<font></font>

08.mi = i<font></font>

09.end<font></font>

10.end<font></font>

11.a.delete_at( mi )<font></font>

12.b << m<font></font>

13.end<font></font>
```

এই পদ্ধতিকে সিলেকশন সর্ট বলা হয় । এটি আমাদের অ্যারের সর্বনিম্ন খুঁজে বের করে (অ্যারেটি উপরে *একটি* চিহ্নিত করা হয় , যখন ন্যূনতম মানটি *m* এবং *mi* বোঝানো হয় এর সূচক), এটি একটি নতুন অ্যারের শেষে রাখে (আমাদের ক্ষেত্রে *b*), এবং এটি থেকে সরিয়ে দেয় মূল অ্যারে। তারপর এটি আমাদের মূল অ্যারের অবশিষ্ট মানগুলির মধ্যে সর্বনিম্ন খুঁজে বের করে, এটিকে আমাদের নতুন অ্যারেতে যুক্ত করে যাতে এটি এখন দুটি উপাদান ধারণ করে এবং এটিকে আমাদের মূল অ্যারে থেকে সরিয়ে দেয়। এটি এই প্রক্রিয়াটি অব্যাহত রাখে যতক্ষণ না সমস্ত আইটেম মূল থেকে সরানো হয় এবং নতুন অ্যারেতে ঢোকানো হয়, যার মানে অ্যারে সাজানো হয়েছে। এই উদাহরণে, আমরা দেখতে পাচ্ছি যে আমাদের দুটি নেস্টেড লুপ রয়েছে। বাইরের লুপ *n* বার চলে, এবং ভিতরের লুপ অ্যারের প্রতিটি উপাদানের জন্য একবার চলে *a* . অ্যারের *প্রাথমিকভাবে n* আইটেম থাকলেও , আমরা প্রতিটি পুনরাবৃত্তিতে একটি অ্যারে আইটেম সরিয়ে ফেলি। সুতরাং অভ্যন্তরীণ লুপটি বাইরের লুপের প্রথম পুনরাবৃত্তির সময় *n*n - 1 বার পুনরাবৃত্তি করে, তারপরে বার, তারপরে n - 2বার ইত্যাদি, বাইরের লুপের শেষ পুনরাবৃত্তি না হওয়া পর্যন্ত, যার সময় এটি শুধুমাত্র একবার চলে।

এই প্রোগ্রামটির জটিলতা মূল্যায়ন করা একটু কঠিন, কারণ আমাদের যোগফল 1 + 2 + ... + (n - 1) + n বের করতে হবে। তবে আমরা নিশ্চিতভাবে এটির জন্য একটি "উপরের সীমা" খুঁজে পেতে পারি। অর্থাৎ, আমরা আমাদের প্রোগ্রামটি পরিবর্তন করতে পারি (আপনি এটি আপনার মনে করতে পারেন, প্রকৃত কোডে নয়) এটিকে এটির চেয়ে **খারাপ** করতে এবং তারপরে আমরা উদ্ভূত নতুন প্রোগ্রামটির জটিলতা খুঁজে বের করতে পারি। যদি আমরা আমাদের তৈরি করা খারাপ প্রোগ্রামটির জটিলতা খুঁজে পেতে পারি, তাহলে আমরা জানি যে আমাদের মূল প্রোগ্রামটি সবচেয়ে বেশি খারাপ, বা আরও ভাল। এইভাবে, যদি আমরা আমাদের পরিবর্তিত প্রোগ্রামের জন্য একটি সুন্দর

জটিলতা খুঁজে বের করি, যা আমাদের মূলের চেয়ে খারাপ, আমরা জানতে পারি যে আমাদের মূল প্রোগ্রামটিও বেশ ভাল জটিলতা থাকবে - হয় আমাদের পরিবর্তিত প্রোগ্রামের মতো বা আরও ভাল।

আসুন এখন এই উদাহরণ প্রোগ্রামটি সম্পাদনা করার উপায় সম্পর্কে চিন্তা করি যাতে এটির জটিলতা বের করা সহজ হয়। তবে আসুন মনে রাখবেন যে আমরা এটিকে আরও খারাপ করতে পারি, অর্থাৎ এটিকে আরও নির্দেশাবলী গ্রহণ করতে, যাতে আমাদের অনুমান আমাদের মূল প্রোগ্রামের জন্য অর্থবহ হয়। স্পষ্টতই আমরা প্রোগ্রামের অভ্যন্তরীণ লুপ পরিবর্তন করতে পারি যাতে বিভিন্ন সময়ের পরিবর্তে সবসময় ঠিক *n* বার পুনরাবৃত্তি করা যায়। এই পুনরাবৃত্তির কিছু অকেজো হবে, কিন্তু এটি আমাদের ফলাফল অ্যালগরিদমের জটিলতা বিশ্লেষণ করতে সাহায্য করবে। যদি আমরা এই সাধারণ পরিবর্তনটি করি, তাহলে আমরা যে নতুন অ্যালগরিদমটি তৈরি করেছি তা স্পষ্টভাবে $\Theta(n^2)$, কারণ আমাদের দুটি নেস্টেড লুপ রয়েছে যেখানে প্রতিটি ঠিক *n* বার পুনরাবৃত্তি হয়। যদি তাই হয়, আমরা বলি যে আসল অ্যালগরিদম হল $O(n^2)$। $O(n^2)$ উচ্চারিত হয় "n বর্গের বড় ওহ"। এটি যা বলে তা হল যে আমাদের প্রোগ্রামটি লক্ষণগতভাবে n^2 এর চেয়ে খারাপ নয় । এটি তার চেয়েও ভাল হতে পারে, বা এটি একই রকম হতে পারে। যাইহোক, যদি আমাদের প্রোগ্রামটি প্রকৃতপক্ষে $\Theta(n^2)$ হয় তবে আমরা এখনও বলতে পারি যে এটি $O(n^2)$। আপনাকে এটি বুঝতে সাহায্য করার জন্য, মূল প্রোগ্রামটিকে এমনভাবে পরিবর্তন করার কল্পনা করুন যা এটিকে খুব বেশি পরিবর্তন করে না, তবে এটিকে আরও খারাপ করে তোলে, যেমন প্রোগ্রামের শুরুতে একটি অর্থহীন নির্দেশ যোগ করা। এটি করা একটি সাধারণ ধ্রুবক দ্বারা নির্দেশ-গণনা ফাংশনকে পরিবর্তন করবে, যা উপেক্ষা করা হয় যখন এটি অ্যাসিম্পোটিক আচরণ আসে। সুতরাং একটি প্রোগ্রাম যা $\Theta(n^2)$ ও $O(n^2)$।

কিন্তু একটি প্রোগ্রাম যা $O(n^{2}) \Theta(n^2)$ নাও হতে পারে । উদাহরণস্বরূপ, যে কোনো প্রোগ্রাম যা $\Theta(n)$ হয় $O(n)$ ছাড়াও $O(n^2)$। আমরা যদি কল্পনা করি যে একটি $\Theta(n)$ প্রোগ্রাম একটি সাধারণ `for`লুপ যা *n* বার পুনরাবৃত্তি হয়, আমরা এটিকে আরও খারাপ করতে পারি অন্য একটি লুপে মোড়ানো `for`যা *n* বারও পুনরাবৃত্তি হয়, এইভাবে $f(n) = n^2$ সহ একটি প্রোগ্রাম তৈরি করা যায় . এটিকে সাধারণীকরণ করার জন্য, যে কোনো প্রোগ্রাম যা $\Theta(a)$ হয় $O(b)$ যখন *b a এর* চেয়ে খারাপ হয় । লক্ষ্য করুন যে প্রোগ্রামে আমাদের পরিবর্তনের জন্য আমাদের এমন একটি প্রোগ্রাম দেওয়ার দরকার নেই যা আসলে অর্থপূর্ণ বা আমাদের মূল প্রোগ্রামের সমতুল্য। এটি শুধুমাত্র একটি প্রদত্ত *n* এর জন্য মূলের চেয়ে বেশি নির্দেশাবলী সম্পাদন করতে হবে । আমরা যা ব্যবহার করছি তা হল নির্দেশনা গণনা, আসলে আমাদের সমস্যার সমাধান নয়।

সুতরাং, আমাদের প্রোগ্রাম $O(n^2)$ বলা নিরাপদ দিকে হচ্ছে: আমরা আমাদের অ্যালগরিদম বিশ্লেষণ করেছি, এবং আমরা দেখেছি যে এটি কখনই n^2 এর চেয়ে খারাপ নয় । কিন্তু এটা হতে পারে যে এটা আসলে n^2 । এটি আমাদের প্রোগ্রামটি কত দ্রুত চলে তার একটি ভাল অনুমান দেয়। এই নতুন স্বরলিপির সাথে নিজেকে পরিচিত করতে সাহায্য করার জন্য আসুন কয়েকটি উদাহরণ দিয়ে যাই।

ব্যায়াম 3

নিচের কোনটি সত্য তা জানুন:

1. একটি $\Theta(n)$ অ্যালগরিদম হল $O(n)$
2. একটি $\Theta(n)$ অ্যালগরিদম হল $O(n^2)$
3. A $\Theta(n^2)$ অ্যালগরিদম হল $O(n^3)$
4. A $\Theta(n)$ অ্যালগরিদম হল $O(1)$

5. AO(1) অ্যালগরিদম হল Θ(1)
6. AO(n) অ্যালগরিদম হল Θ(1)

সমাধান

1. আমরা জানি যে এটি সত্য কারণ আমাদের মূল প্রোগ্রাম ছিল Θ(n)। আমরা আমাদের প্রোগ্রামকে একেবারে পরিবর্তন না করেই O(n) অর্জন করতে পারি।
2. যেহেতু n^2 n এর চেয়ে খারাপ, এটি সত্য।
3. যেহেতু n^{3n^2} এর চেয়ে খারাপ , এটি সত্য।
4. যেহেতু 1 n এর চেয়ে খারাপ নয়, এটি মিথ্যা। যদি একটি প্রোগ্রাম *n* নির্দেশাবলী অসংলগ্নভাবে গ্রহণ করে (নির্দেশের একটি রৈখিক সংখ্যা), আমরা এটিকে আরও খারাপ করতে পারি না এবং এটিকে শুধুমাত্র 1টি নির্দেশনা অচিহ্নিতভাবে নিতে হবে (নির্দেশের একটি ধ্রুবক সংখ্যা)।
5. এটি সত্য কারণ দুটি জটিলতা একই।
6. অ্যালগরিদমের উপর নির্ভর করে এটি সত্য হতে পারে বা নাও হতে পারে। সাধারণ ক্ষেত্রে এটি মিথ্যা। যদি একটি অ্যালগরিদম হয় Θ(1), তাহলে অবশ্যই এটি O(n)। কিন্তু যদি এটি O(n) হয় তবে এটি Θ(1) নাও হতে পারে। উদাহরণস্বরূপ, একটি Θ(n) অ্যালগরিদম হল O(n) কিন্তু Θ(1) নয়।

ব্যায়াম 4

একটি গাণিতিক অগ্রগতি যোগফল ব্যবহার করে প্রমাণ করুন যে উপরের প্রোগ্রামটি শুধুমাত্র $O(n^{2})$ নয়, $\Theta(n^2)$ ও । আপনি যদি না জানেন যে একটি গাণিতিক অগ্রগতি কি, উইকিপিডিয়াতে দেখুন - এটি সহজ।

কারণ একটি অ্যালগরিদমের O-জটিলতা একটি অ্যালগরিদমের প্রকৃত জটিলতার জন্য একটি *উপরের সীমানা* দেয় , যখন Θ একটি অ্যালগরিদমের প্রকৃত জটিলতা দেয়, আমরা কখনও কখনও বলি যে Θ আমাদের একটি *আঁটসাঁট বাঁধা* দেয় । যদি আমরা জানি যে আমরা এমন একটি জটিলতা খুঁজে পেয়েছি যা আঁটসাঁট নয়, আমরা এটি বোঝাতে একটি ছোট হাতের অক্ষরও ব্যবহার করতে পারি। উদাহরণস্বরূপ, যদি একটি অ্যালগরিদম Θ(n) হয়, তাহলে এর শক্ত জটিলতা হল n। তাহলে এই অ্যালগরিদমটি হল O(n) এবং $O(n^2)$ উভয়ই। যেহেতু অ্যালগরিদম হল Θ(n), তাই O(n) আবদ্ধ একটি টাইট। কিন্তু $O(n^2)$ আবদ্ধ নয়, এবং তাই আমরা লিখতে পারি যে অ্যালগরিদম হল $o(n^2)$, যাকে "n স্কোয়ারের ছোট o" উচ্চারণ করা হয় তা বোঝানোর জন্য যে আমরা জানি যে আমাদের আবদ্ধটি টাইট নয়। আমরা যদি আমাদের অ্যালগরিদমগুলির জন্য শক্ত সীমা খুঁজে পেতে পারি তবে এটি আরও ভাল, কারণ এটি আমাদের অ্যালগরিদম কীভাবে আচরণ করে সে সম্পর্কে আরও তথ্য দেয়, তবে এটি করা সবসময় সহজ নয়।

ব্যায়াম 5

নিচের সীমার মধ্যে কোনটি টাইট বাউন্ড এবং কোনটি টাইট বাউন্ড নয় তা নির্ধারণ করুন। কোন সীমা ভুল হতে পারে কিনা দেখতে পরীক্ষা করুন. আঁটসাঁট নয় এমন সীমাগুলিকে চিত্রিত করতে o(স্বরলিপি) ব্যবহার করুন।

1. একটি Θ(n) অ্যালগরিদম যার জন্য আমরা একটি O(n) উপরের বাউন্ড পেয়েছি।
2. একটি $\Theta(n^2)$ অ্যালগরিদম যার জন্য আমরা একটি $O(n^3)$ উপরের বাউন্ড পেয়েছি।
3. একটি Θ(1) অ্যালগরিদম যার জন্য আমরা একটি O(n) উপরের বাউন্ড পেয়েছি।
4. একটি Θ(n) অ্যালগরিদম যার জন্য আমরা একটি O(1) উপরের বাউন্ড পেয়েছি।
5. একটি Θ(n) অ্যালগরিদম যার জন্য আমরা একটি O(2n) উপরের বাউন্ড খুঁজে পেয়েছি।

সমাধান

1. এই ক্ষেত্রে, Θ জটিলতা এবং O জটিলতা একই, তাই আবদ্ধ টাইট।
2. এখানে আমরা দেখতে পাচ্ছি যে O জটিলতা Θ কমপ্লেক্সিটির চেয়ে বৃহত্তর স্কেলের তাই এই আবদ্ধ নয়। প্রকৃতপক্ষে, $O(n^2)$ এর একটি আবদ্ধ একটি আঁটসাঁট হবে। তাই আমরা লিখতে পারি যে অ্যালগরিদম হল $o(n^3)$।
3. আবার আমরা দেখতে পাই যে O জটিলতা Θ জটিলতার চেয়ে বড় আকারের তাই আমাদের একটি আবদ্ধ আছে যা আঁটসাঁট নয়। O(1) এর একটি আবদ্ধ হবে একটি টাইট। সুতরাং আমরা নির্দেশ করতে পারি যে $O(n)$ আবদ্ধ এটিকে $o(n)$ লিখে টাইট নয়।
4. আমরা অবশ্যই এই আবদ্ধ গণনা করতে ভুল করেছি, কারণ এটি ভুল। একটি $\Theta(n)$ অ্যালগরিদমের পক্ষে O(1) এর উপরের সীমা থাকা অসম্ভব, কারণ n হল 1 এর চেয়ে বড় জটিলতা। মনে রাখবেন যে O একটি উপরের সীমা দেয়।
5. এটি আঁটসাঁট নয় এমন একটি আবদ্ধের মতো মনে হতে পারে, তবে এটি আসলে সত্য নয়। এই আবদ্ধ আসলে টাইট. মনে রাখবেন যে 2n এবং n-এর অ্যাসিম্পোটিক আচরণ একই, এবং O এবং Θ শুধুমাত্র অ্যাসিম্পোটিক আচরণের সাথে সম্পর্কিত। সুতরাং আমাদের আছে যে $O(2n) = O(n)$ এবং তাই এই আবদ্ধটি টাইট কারণ জটিলতা Θ এর মতই।

অঙ্গুষ্ঠের নিয়ম : একটি অ্যালগরিদমের O-জটিলতা বের করা তার Θ-জটিলতার চেয়ে সহজ।

আপনি এখন এই সমস্ত নতুন স্বরলিপিতে কিছুটা অভিভূত হতে পারেন, তবে আমরা কয়েকটি উদাহরণে যাওয়ার আগে আরও দুটি প্রতীক পরিচয় করিয়ে দিই। এগুলি এখন সহজ যে আপনি Θ, O এবং o জানেন, এবং আমরা এই নিবন্ধে সেগুলি খুব বেশি পরে ব্যবহার করব না, তবে এখন সেগুলি জেনে নেওয়া ভাল যে আমরা এটিতে আছি। উপরের উদাহরণে, আমরা আমাদের প্রোগ্রামটিকে আরও খারাপ করার জন্য পরিবর্তন করেছি (অর্থাৎ আরও নির্দেশাবলী গ্রহণ করা এবং তাই আরও সময়) এবং O নোটেশন তৈরি করেছি। O অর্থবহ কারণ এটি আমাদের বলে যে আমাদের প্রোগ্রাম কখনই একটি নির্দিষ্ট সীমার চেয়ে ধীর হবে না, এবং তাই এটি মূল্যবান তথ্য প্রদান করে যাতে আমরা যুক্তি দিতে পারি যে আমাদের প্রোগ্রাম যথেষ্ট ভাল। যদি আমরা বিপরীতটি করি এবং আমাদের প্রোগ্রামটিকে আরও **ভাল** করার জন্য পরিবর্তন করি এবং ফলাফলের প্রোগ্রামটির জটিলতা খুঁজে বের করি, আমরা স্বরলিপি Ω ব্যবহার করি। Ω তাই আমাদের একটি জটিলতা দেয় যে আমরা জানি আমাদের প্রোগ্রাম এর চেয়ে ভাল হবে না। এটি দরকারী যদি আমরা প্রমাণ করতে চাই যে একটি প্রোগ্রাম ধীরে ধীরে চলে বা একটি অ্যালগরিদম একটি খারাপ। এটি যুক্তি দিতে উপযোগী হতে পারে যে একটি অ্যালগরিদম একটি নির্দিষ্ট ক্ষেত্রে ব্যবহার করার জন্য খুব ধীর। উদাহরণস্বরূপ, একটি অ্যালগরিদম হল $\Omega(n^3)$ বলার অর্থ হল অ্যালগরিদমটি n^3 এর চেয়ে ভাল নয় । এটি $\Theta(n^3)$, $\Theta(n^4)$ এর মতো খারাপ বা আরও খারাপ হতে পারে , তবে আমরা জানি এটি অন্তত কিছুটা খারাপ। তাই Ω আমাদের অ্যালগরিদমের জটিলতার জন্য একটি *নিম্ন সীমা* দেয় । একইভাবে o, আমরা লিখতে পারি ω যদি আমরা জানি যে আমাদের আবদ্ধ টাইট নয়। উদাহরণস্বরূপ, একটি $\Theta(n^3)$ অ্যালগরিদম হল $o(n^4)$ এবং $\omega(n^2)$। $\Omega(n)$ উচ্চারিত হয় "n এর বড় ওমেগা", যখন $\omega(n)$ উচ্চারিত হয় "n এর ছোট ওমেগা"।

ব্যায়াম 6

নিম্নলিখিত Θ জটিলতার জন্য একটি টাইট এবং একটি নন-টাইট O বাউন্ড, এবং আপনার পছন্দের একটি টাইট এবং অ-টাইট Ω আবদ্ধ লিখুন, যদি সেগুলি বিদ্যমান থাকে।

1. Θ(1)

2. Θ($\sqrt{n}$)
3. Θ(n)
4. Θ(n ^{2})
5. Θ(n ^{3})

সমাধান

এটি উপরের সংজ্ঞাগুলির একটি সোজা-সামনের প্রয়োগ।

1. টাইট বাউন্ড হবে O(1) এবং Ω(1)। একটি নন-টাইট O-বাউন্ড হবে O(n)। স্মরণ করুন যে হে আমাদের একটি উপরের বাউন্ড দেয়। যেহেতু n 1 এর চেয়ে বড় স্কেল এর এটি একটি নন-টাইট বাউন্ড এবং আমরা এটিকে o(n) হিসাবেও লিখতে পারি। কিন্তু আমরা Ω-এর জন্য একটি নন-টাইট বাউন্ড খুঁজে পাচ্ছি না, কারণ আমরা এই ফাংশনের জন্য 1 এর চেয়ে কম পেতে পারি না। তাই আমরা আঁট আবদ্ধ সঙ্গে করতে হবে.
2. আঁটসাঁট সীমাগুলি Θ জটিলতার সমান হতে হবে, তাই তারা যথাক্রমে O($\sqrt{n}$) এবং Ω( )। $\sqrt{n}$নন-টাইট বাউন্ডের জন্য আমাদের O(n) থাকতে পারে, যেহেতু n এর থেকে বড় $\sqrt{n}$এবং তাই এটি একটি উপরের সীমানা $\sqrt{n}$। যেহেতু আমরা জানি এটি একটি নন-টাইট আপার বাউন্ড, আমরা এটিকে o(n) হিসেবেও লিখতে পারি। একটি নিম্ন সীমার জন্য যা আঁটসাঁট নয়, আমরা কেবল Ω(1) ব্যবহার করতে পারি। যেহেতু আমরা জানি যে এই আবদ্ধ টাইট নয়, আমরা এটিকে ω(1) হিসাবেও লিখতে পারি।
3. টাইট বাউন্ডগুলি হল O(n) এবং Ω(n)। দুটি নন-টাইট বাউন্ড ω(1) এবং o(n ^{3}) হতে পারে । এগুলি আসলে বেশ খারাপ সীমা, কারণ এগুলি মূল জটিলতা থেকে অনেক দূরে, কিন্তু আমাদের সংজ্ঞাগুলি ব্যবহার করে সেগুলি এখনও বৈধ৷
4. টাইট বাউন্ডগুলি হল O(n ^{2}) এবং Ω(n ^{2})। নন-টাইট বাউন্ডের জন্য আমরা আবার ω(1) এবং o(n ^{3}) ব্যবহার করতে পারি আমাদের আগের উদাহরণের মতো।
5. টাইট বাউন্ডগুলি হল যথাক্রমে O(n ^{3}) এবং Ω(n $^3\sqrt{n}$)। দুটি নন-টাইট বাউন্ড ω(n ^{2}) এবং o($\sqrt{n}$n ^{3}) হতে পারে । যদিও এই সীমাবদ্ধতাগুলি আঁটসাঁট নয়, আমরা উপরে যেগুলি দিয়েছি তার চেয়ে এগুলি ভাল।

যে কারণে আমরা Θ-এর পরিবর্তে O এবং Ω ব্যবহার করি যদিও O এবং Ω টাইট বাউন্ডও দিতে পারে তা হল যে আমরা হয়তো বলতে পারব না যে আমরা খুঁজে পেয়েছি একটি বাউন্ড টাইট কিনা, অথবা আমরা হয়তো প্রক্রিয়াটির মধ্য দিয়ে যেতে চাই না এটা এত যাচাই করে.

আপনি যদি সমস্ত ভিন্ন চিহ্ন এবং তাদের ব্যবহারগুলি সম্পূর্ণরূপে মনে না রাখেন তবে এখনই এটি সম্পর্কে খুব বেশি চিন্তা করবেন না। আপনি সবসময় ফিরে আসতে পারেন এবং তাদের দেখতে পারেন. সবচেয়ে গুরুত্বপূর্ণ চিহ্ন হল O এবং Θ।

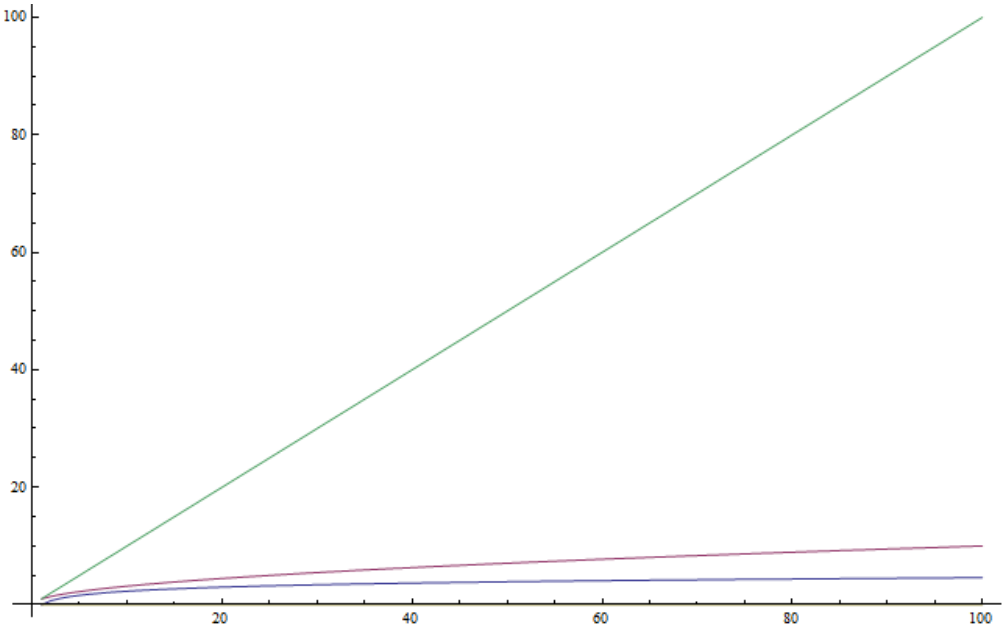
এছাড়াও মনে রাখবেন যে যদিও Ω আমাদের ফাংশনের জন্য একটি নিম্ন-বাউন্ড আচরণ দেয় (অর্থাৎ আমরা আমাদের প্রোগ্রামটি উন্নত করেছি এবং এটি কম নির্দেশাবলী সম্পাদন করেছি) আমরা এখনও একটি "সবচেয়ে খারাপ-কেস" বিশ্লেষণের কথা উল্লেখ করছি। আমরা আমাদের প্রোগ্রাম ফিড করছি কারণ একটি প্রদত্ত n জন্য সবচেয়ে খারাপ সম্ভাব্য ইনপুট এবং এই অনুমান অধীনে তার আচরণ বিশ্লেষণ.

নিম্নলিখিত সারণীটি আমরা এইমাত্র প্রবর্তিত চিহ্নগুলি এবং তুলনার সাধারণ গাণিতিক চিহ্নগুলির সাথে তাদের সঙ্গতি নির্দেশ করে যা আমরা সংখ্যার জন্য ব্যবহার করি। আমরা এখানে সাধারণ চিহ্নগুলি ব্যবহার না করার এবং পরিবর্তে

গ্রীক অক্ষর ব্যবহার করার কারণ হল যে আমরা একটি অ্যাসিম্পোটিক আচরণ তুলনা করছি, কেবল একটি সাধারণ তুলনা নয়।

অ্যাসিম্পটোটিক তুলনা অপারেটর	**সংখ্যাসূচক তুলনা অপারেটর**
আমাদের অ্যালগরিদম হল **o** (কিছু)	একটি সংখ্যা হল **<** কিছু
আমাদের অ্যালগরিদম হল **O** (কিছু)	একটি সংখ্যা হল **≤** কিছু
আমাদের অ্যালগরিদম হল **Θ** (কিছু)	একটি সংখ্যা **=** কিছু
আমাদের অ্যালগরিদম হল **Ω** (কিছু)	একটি সংখ্যা হল **≥** কিছু
আমাদের অ্যালগরিদম হল **ω** (কিছু)	একটি সংখ্যা হল **>** কিছু

অঙ্গুষ্ঠের নিয়ম : O, o, Ω, ω এবং Θ সবকটি চিহ্নই অনেক সময় কাজে লাগে, O হল বেশি ব্যবহৃত হয়, কারণ এটি Θ-এর চেয়ে নির্ণয় করা সহজ এবং Ω-এর চেয়ে ব্যবহারিকভাবে বেশি কার্যকর।



লগারিদম

লগারিদম কি তা যদি আপনি জানেন, তাহলে নির্দ্বিধায় এই বিভাগটি এড়িয়ে যান। যেহেতু অনেক লোক লগারিদমগুলির সাথে অপরিচিত, বা সম্প্রতি সেগুলি খুব বেশি ব্যবহার করেনি এবং সেগুলি মনে রাখে না, এই বিভাগটি এখানে তাদের জন্য একটি ভূমিকা হিসাবে রয়েছে। এই পাঠ্যটি অল্প বয়স্ক ছাত্রদের জন্যও যারা এখনও স্কুলে লগারিদম দেখেননি। লগারিদমগুলি গুরুত্বপূর্ণ কারণ জটিলতা বিশ্লেষণ করার সময় তারা অনেক বেশি ঘটে। লগারিদম হল একটি সংখ্যার উপর প্রয়োগ করা একটি অপারেশন যা এটিকে বেশ ছোট করে তোলে - অনেকটা একটি সংখ্যার বর্গমূলের মতো / সুতরাং লগারিদম সম্পর্কে আপনি যদি একটি জিনিস মনে রাখতে চান তা হল তারা একটি সংখ্যা নেয় এবং এটিকে আসল থেকে অনেক ছোট করে (**চিত্র 4** দেখুন)। এখন, যেভাবে বর্গমূল হল কোনো কিছুর বর্গ করার বিপরীত ক্রিয়াকলাপ, লগারিদম হল কোনো কিছুর সূচকের বিপরীত ক্রিয়া। এটি যতটা কঠিন মনে হচ্ছে ততটা কঠিন নয়। এটি একটি উদাহরণ দিয়ে আরও ভালভাবে ব্যাখ্যা করা হয়েছে। সমীকরণ বিবেচনা করুন:

$2^x = 1024$

আমরা এখন x এর জন্য এই সমীকরণটি সমাধান করতে চাই । তাই আমরা নিজেদেরকে জিজ্ঞাসা করি: কোন সংখ্যার জন্য আমাদের বেস 2 বাড়াতে হবে যাতে আমরা 1024 পেতে পারি? সেই সংখ্যাটি হল 10। প্রকৃতপক্ষে, আমাদের কাছে $2^{10} = 1024$ আছে, যা যাচাই করা সহজ। লগারিদম আমাদের নতুন স্বরলিপি ব্যবহার করে এই সমস্যাটি বোঝাতে সাহায্য করে। এই ক্ষেত্রে, 10 হল 1024 এর লগারিদম এবং আমরা এটিকে log(1024) হিসাবে লিখি এবং আমরা এটিকে "1024 এর লগারিদম" হিসাবে পড়ি। যেহেতু আমরা 2 কে বেস হিসাবে ব্যবহার করছি, এই লগারিদমগুলিকে বেস 2 লগারিদম বলা হয়। অন্যান্য বেসে লগারিদম আছে, কিন্তু আমরা এই নিবন্ধে শুধুমাত্র বেস 2 লগারিদম ব্যবহার করব। আপনি যদি আন্তর্জাতিক প্রতিযোগিতায় অংশগ্রহণকারী একজন ছাত্র হন এবং লগারিদম সম্পর্কে আপনি জানেন না, আমি অত্যন্ত সুপারিশ করছি যে আপনি এই নিবন্ধটি সম্পূর্ণ করার পরে আপনার লগারিদম অনুশীলন করুন । কম্পিউটার বিজ্ঞানে, বেস 2 লগারিদম অন্য যেকোনো ধরনের লগারিদমের তুলনায় অনেক বেশি সাধারণ। এর কারণ হল আমাদের প্রায়শই শুধুমাত্র দুটি ভিন্ন সত্তা থাকে: 0 এবং 1। এছাড়াও আমরা একটি বড় সমস্যাকে অর্ধেক করে ফেলি, যার মধ্যে সবসময় দুটি থাকে। তাই এই নিবন্ধটি চালিয়ে যাওয়ার জন্য আপনাকে শুধুমাত্র বেস-2 লগারিদম সম্পর্কে জানতে হবে।

ব্যায়াম 7

নিচের সমীকরণগুলো সমাধান কর। প্রতিটি ক্ষেত্রে আপনি কোন লগারিদম খুঁজে পাচ্ছেন তা নির্দেশ করুন। শুধুমাত্র লগারিদম বেস 2 ব্যবহার করুন।

1. $2^x = 64$
2. $(2^2)^x = 64$
3. $4^x = 4$
4. $2^x = 1$
5. $2^x + 2^x = 32$
6. $(2^x) * (2^x) = 64$

Solution

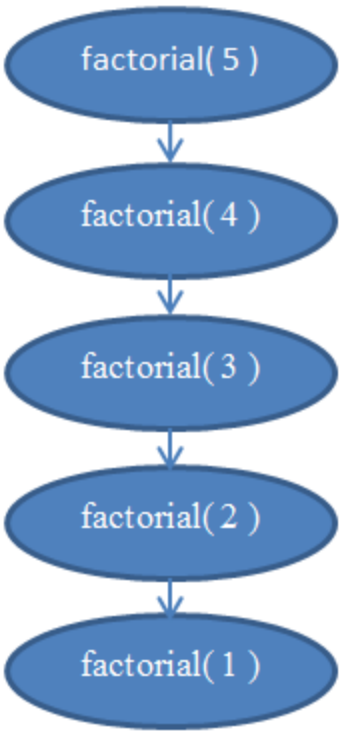
There is nothing more to this than applying the ideas defined above.

1. By trial and error we can find that x = 6 and so log(64) = 6.
2. Here we notice that $(2^2)^x$, by the properties of exponents, can be written as 2^{2x}. So we have that 2x = 6 because log(64) = 6 from the previous result and therefore x = 3.
3. Using our knowledge from the previous equation, we can write 4 as 2^2 and so our equation becomes $(2^2)^x = 4$ which is the same as $2^{2x} = 4$. Then we notice that log(4) = 2 because $2^2 = 4$ and therefore we have that 2x = 2. So x = 1. This is readily observed from the original equation, as using an exponent of 1 yields the base as a result.
4. Recall that an exponent of 0 yields a result of 1. So we have log(1) = 0 as $2^0 = 1$, and so x = 0.
5. Here we have a sum and so we can't take the logarithm directly. However we notice that $2^x + 2^x$ is the same as $2 * (2^x)$. So we've multiplied in yet another two, and therefore this is the same as $2^{x + 1}$ and

now all we have to do is solve the equation $2^{x+1} = 32$. We find that log(32) = 5 and so x + 1 = 5 and therefore x = 4.

6. We're multiplying together two powers of 2, and so we can join them by noticing that $(2^x) * (2^x)$ is the same as 2^{2x}. Then all we need to do is to solve the equation $2^{2x} = 64$ which we already solved above and so x = 3.

Rule of thumb: For competition algorithms implemented in C++, once you've analyzed your complexity, you can get a rough estimate of how fast your program will run by expecting it to perform about 1,000,000 operations per second, where the operations you count are given by the asymptotic behavior function describing your algorithm. For example, a Θ(n) algorithm takes about a second to process the input for n = 1,000,000.



Recursive complexity

Let's now take a look at a recursive function. A *recursive function* is a function that calls itself. Can we analyze its complexity? The following function, written in Python, evaluates the factorial of a given number. The factorial of a positive integer number is found by multiplying it with all the previous positive integers together. For example, the factorial of 5 is 5 * 4 * 3 * 2 * 1. We denote that "5!" and pronounce it "five factorial" (some people prefer to pronounce it by screaming it out aloud like "FIVE!!!")

```
1.def factorial( n ):<font></font>

2.if n == 1:<font></font>

3.return 1<font></font>
```

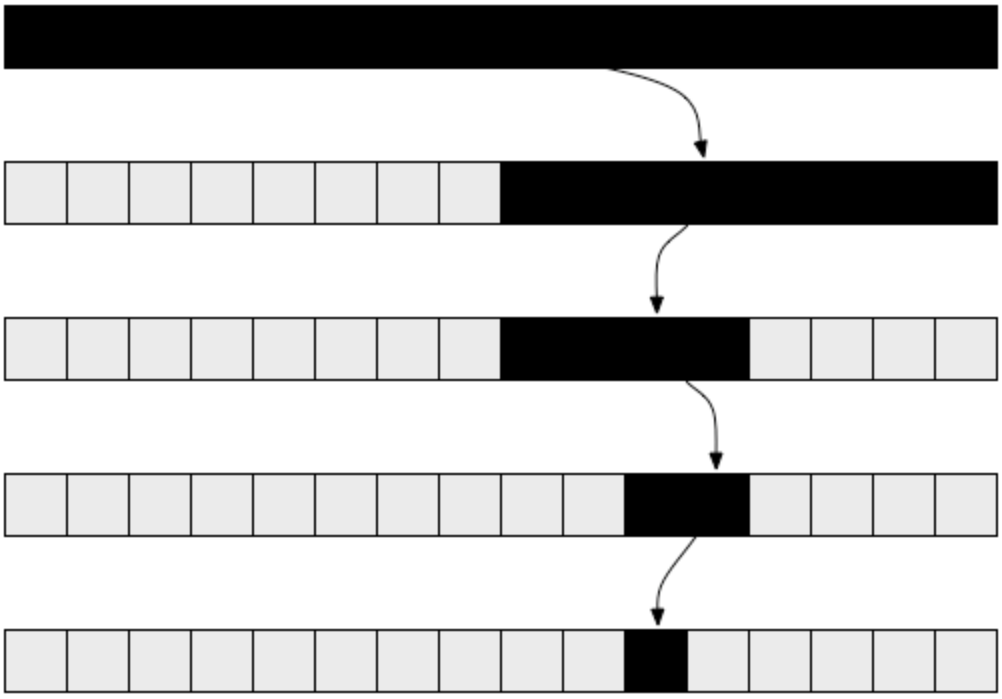
```
4.return n * factorial( n - 1 )<font></font>
```

আসুন এই ফাংশনের জটিলতা বিশ্লেষণ করি। এই ফাংশনটিতে কোনও লুপ নেই, তবে এর জটিলতাও স্থির নয়। এর জটিলতা খুঁজে বের করার জন্য আমাদের যা করতে হবে তা হল আবার নির্দেশনা গণনা করা। স্পষ্টতই, যদি আমরা এই ফাংশনে কিছু *n পাস করি, এটি নিজেই n* বার চালাবে। আপনি যদি এটি সম্পর্কে অনিশ্চিত হন তবে এটি আসলে কাজ করে কিনা তা যাচাই করতে n = 5 এর জন্য এখন এটি "হাত দ্বারা" চালান। উদাহরণস্বরূপ, n = 5 এর জন্য, এটি 5 বার কার্যকর করবে, কারণ এটি প্রতিটি কলে 1 দ্বারা n কমতে থাকবে। তাই আমরা দেখতে পাচ্ছি যে এই ফাংশনটি তখন Θ(n)।

আপনি যদি এই সত্য সম্পর্কে অনিশ্চিত হন তবে মনে রাখবেন যে আপনি নির্দেশাবলী গণনা করে সর্বদা সঠিক জটিলতা খুঁজে পেতে পারেন। আপনি যদি চান, আপনি এখন একটি ফাংশন f(n) খুঁজে পেতে এই ফাংশন দ্বারা সম্পাদিত প্রকৃত নির্দেশগুলি গণনা করার চেষ্টা করতে পারেন এবং দেখতে পারেন যে এটি প্রকৃতপক্ষে রৈখিক (মনে করুন সেই লিনিয়ার মানে Θ(n))।

ফ্যাক্টরিয়াল(5) বলা হলে সঞ্চালিত পুনরাবৃত্তি বুঝতে সাহায্য করার জন্য একটি ডায়াগ্রামের জন্য **চিত্র 5** দেখুন ।

কেন এই ফাংশন রৈখিক জটিলতা এর এটি পরিষ্কার করা উচিত।



লগারিদমিক জটিলতা

কম্পিউটার বিজ্ঞানের একটি বিখ্যাত সমস্যা হল একটি অ্যারের মধ্যে একটি মান অনুসন্ধান করা। আমরা সাধারণ ক্ষেত্রে এই সমস্যাটি আগে সমাধান করেছি। এই সমস্যাটি আকর্ষণীয় হয়ে ওঠে যদি আমাদের একটি অ্যারে থাকে যা সাজানো থাকে এবং আমরা এটির মধ্যে একটি প্রদত্ত মান খুঁজে পেতে চাই। এটি করার একটি পদ্ধতি *বাইনারি অনুসন্ধান* বলা হয় । আমরা আমাদের অ্যারের মধ্যম উপাদান তাকান: যদি আমরা এটি সেখানে খুঁজে পাই, আমরা সম্পন্ন করেছি। অন্যথায়, যদি আমরা সেখানে যে মানটি খুঁজে পাই তা আমরা যে মানটি খুঁজছি তার চেয়ে বড় হয়, আমরা জানি যে আমাদের উপাদানটি অ্যারের বাম অংশে থাকবে। অন্যথায়, আমরা জানি এটি অ্যারের ডান অংশে

থাকবে। আমরা এই ছোট অ্যারেগুলিকে অর্ধেক করে কেটে রাখতে পারি যতক্ষণ না আমাদের দেখার জন্য একটি একক উপাদান থাকে। এখানে pseudocode ব্যবহার করে পদ্ধতি:

```
01.def binarySearch( A, n, value ):<font></font>

02.if n = 1:<font></font>

03.if A[ 0 ] = value:<font></font>

04.return true<font></font>

05.else:<font></font>

06.return false<font></font>

07.if value < A[ n / 2 ]:<font></font>

08.return binarySearch( A[ 0...( n / 2 - 1 ) ], n / 2 - 1, value )<font></font>

09.else if value > A[ n / 2 ]:<font></font>

10.return binarySearch( A[ ( n / 2 + 1 )...n ], n / 2 - 1, value )<font></font>

11.else:<font></font>

12.return true<font></font>
```

এই pseudocode প্রকৃত বাস্তবায়ন একটি সরলীকরণ. অনুশীলনে, এই পদ্ধতিটি বাস্তবায়নের চেয়ে সহজ বর্ণনা করা হয়েছে, কারণ প্রোগ্রামারকে কিছু বাস্তবায়নের সমস্যাগুলির যত্ন নেওয়া দরকার। একের পর এক ত্রুটি রয়েছে এবং 2 দ্বারা বিভাজন সর্বদা একটি পূর্ণসংখ্যার মান তৈরি করতে পারে না এবং তাই ফ্লোর() বা সিল() মানটি প্রয়োজনীয়। কিন্তু আমরা আমাদের উদ্দেশ্যের জন্য অনুমান করতে পারি যে এটি সর্বদা সফল হবে, এবং আমরা অনুমান করব যে আমাদের বাস্তব বাস্তবায়ন বাস্তবে একের পর এক ত্রুটির যত্ন নেয়, কারণ আমরা শুধুমাত্র এই পদ্ধতির জটিলতা বিশ্লেষণ করতে চাই। আপনি যদি আগে কখনও বাইনারি অনুসন্ধান প্রয়োগ না করে থাকেন তবে আপনি এটি আপনার প্রিয় প্রোগ্রামিং ভাষায় করতে চাইতে পারেন। এটা সত্যিই একটি আলোকিত প্রচেষ্টা.

বাইনারি অনুসন্ধান কীভাবে কাজ করে তা বুঝতে আপনাকে সাহায্য করতে **চিত্র 6** দেখুন ।

আপনি যদি নিশ্চিত না হন যে এই পদ্ধতিটি আসলে কাজ করে, এখন একটি মুহূর্ত সময় নিন একটি সাধারণ উদাহরণে এটি হাতে চালানোর জন্য এবং নিজেকে বোঝান যে এটি আসলে কাজ করে।

আসুন এখন এই অ্যালগরিদম বিশ্লেষণ করার চেষ্টা করি। আবার, এই ক্ষেত্রে আমাদের একটি পুনরাবৃত্ত অ্যালগরিদম আছে। ধরা যাক, সরলতার জন্য, যে অ্যারেটি সর্বদা ঠিক অর্ধেক কাটা হয়, এখনই রিকার্সিভ কলে + 1 এবং - 1 অংশটিকে উপেক্ষা করে। এখন পর্যন্ত আপনার নিশ্চিত হওয়া উচিত যে সামান্য পরিবর্তন যেমন + 1 এবং - 1 উপেক্ষা করা আমাদের জটিলতার ফলাফলকে প্রভাবিত করবে না। এটি একটি সত্য যে আমরা গাণিতিক দৃষ্টিকোণ থেকে বিচক্ষণ হতে চাইলে আমাদের সাধারণত প্রমাণ করতে হবে, কিন্তু কার্যত এটি স্বজ্ঞাতভাবে সুস্পষ্ট। ধরা যাক যে আমাদের অ্যারের একটি আকার রয়েছে যা সরলতার জন্য 2 এর সঠিক শক্তি। আবার এই অনুমান আমাদের জটিলতার চূড়ান্ত ফলাফলকে পরিবর্তন করে না যা আমরা পৌঁছাব। এই সমস্যার জন্য সবচেয়ে খারাপ পরিস্থিতি

ঘটবে যখন আমরা যে মানটি খুঁজছি তা আমাদের অ্যারেতে ঘটবে না। সেক্ষেত্রে, আমরা পুনরাবৃত্তির প্রথম কলে n আকারের অ্যারে দিয়ে শুরু করব, তারপর পরবর্তী কলে n/2 আকারের অ্যারে পাব। তারপর আমরা পরবর্তী রিকার্সিভ কলে সাইজ n/4 এর একটি অ্যারে পাব, তারপরে n/8 আকারের অ্যারে এবং আরও অনেক কিছু। সাধারণভাবে, আমাদের অ্যারে প্রতিটি কলে অর্ধেক ভাগে বিভক্ত হয়, যতক্ষণ না আমরা 1 এ পৌঁছাই। সুতরাং, আসুন প্রতিটি কলের জন্য আমাদের অ্যারেতে উপাদানের সংখ্যা লিখি:

1. 0 ম পুনরাবৃত্তি: n
2. 1ম পুনরাবৃত্তি : n/2
3. ২ য় পুনরাবৃত্তি: n/4
4. 3 য় পুনরাবৃত্তি: n/8
5. ...
6. i th পুনরাবৃত্তি: $n / 2^i$
7. ...
8. শেষ পুনরাবৃত্তি: 1

লক্ষ্য করুন যে i-th পুনরাবৃত্তিতে, আমাদের অ্যারেতে $n/2^i$ উপাদান রয়েছে। এর কারণ হল প্রতিটি পুনরাবৃত্তিতে আমরা আমাদের অ্যারেকে অর্ধেক করে কাটছি, যার অর্থ আমরা এর উপাদানগুলির সংখ্যাকে দুই দ্বারা ভাগ করছি। এটি একটি 2 দিয়ে হরকে গুন করতে অনুবাদ করে। যদি আমরা i বার করি, আমরা $n / 2^i$ পাব । এখন, এই পদ্ধতিটি চলতে থাকে এবং প্রতিটি বড় i এর সাথে আমরা একটি ছোট সংখ্যক উপাদান পাই যতক্ষণ না আমরা শেষ পুনরাবৃত্তিতে না পৌঁছাই যেখানে আমাদের কেবল 1টি উপাদান অবশিষ্ট রয়েছে। যদি আমরা i খুঁজে পেতে চাই যে এটি কোন পুনরাবৃত্তিতে ঘটবে, আমাদের নিম্নলিখিত সমীকরণটি সমাধান করতে হবে:

$1 = n/2^i$

এটি তখনই সত্য হবে যখন আমরা বাইনারি সার্চ() ফাংশনে চূড়ান্ত কলে পৌঁছে যাই, সাধারণ ক্ষেত্রে নয়। তাই এখানে i-এর সমাধান করা আমাদের খুঁজে পেতে সাহায্য করবে কোন পুনরাবৃত্তিতে পুনরাবৃত্তি শেষ হবে। উভয় পক্ষকে 2 দ্বারা গুণ করলে আমরা পাই:

$2^i = n$

এখন, আপনি উপরের লগারিদম বিভাগটি পড়লে এই সমীকরণটি পরিচিত দেখা উচিত। আমার জন্য সমাধান করা আমাদের আছে:

$i = \log(n)$

এটি আমাদের বলে যে একটি বাইনারি অনুসন্ধান করার জন্য প্রয়োজনীয় পুনরাবৃত্তির সংখ্যা হল log(n) যেখানে n হল মূল অ্যারের উপাদানগুলির সংখ্যা।

আপনি যদি এটি সম্পর্কে চিন্তা করেন তবে এটি কিছুটা অর্থবহ। উদাহরণস্বরূপ, n = 32 নিন, 32টি উপাদানের একটি অ্যারে। মাত্র 1টি উপাদান পেতে আমাদের কতবার এটিকে অর্ধেক করতে হবে? আমরা পাই: 32 → 16 → 8 → 4 → 2 → 1. আমরা এটি 5 বার করেছি, যা 32 এর লগারিদম। অতএব, বাইনারি অনুসন্ধানের জটিলতা হল $\Theta(\log(n))$।

এই শেষ ফলাফলটি আমাদের পূর্ববর্তী পদ্ধতি, লিনিয়ার অনুসন্ধানের সাথে বাইনারি অনুসন্ধানের তুলনা করতে দেয়। স্পষ্টতই, যেহেতু log(n) n এর চেয়ে অনেক ছোট, তাই এই উপসংহারে আসা যুক্তিসঙ্গত যে লিনিয়ার সার্চের চেয়ে অ্যারের মধ্যে অনুসন্ধান করার জন্য বাইনারি অনুসন্ধান একটি অনেক দ্রুত পদ্ধতি, তাই আমরা যদি করতে চাই তবে আমাদের অ্যারেগুলিকে সাজানো রাখার পরামর্শ দেওয়া যেতে পারে। তাদের মধ্যে অনেক অনুসন্ধান.

অঙ্গুষ্ঠের নিয়ম : একটি প্রোগ্রামের অ্যাসিম্পোটিক চলমান সময়কে উন্নত করা প্রায়শই এর কার্যকারিতা ব্যাপকভাবে বৃদ্ধি করে, যে কোনও ছোট "প্রযুক্তিগত" অপ্টিমাইজেশন যেমন একটি দ্রুত প্রোগ্রামিং ভাষা ব্যবহার করার চেয়ে অনেক বেশি।

সর্বোত্তম বাছাই

অভিনন্দন। আপনি এখন অ্যালগরিদমের জটিলতা, ফাংশনের অ্যাসিম্পোটিক আচরণ এবং বিগ-ও নোটেশনের বিশ্লেষণ সম্পর্কে জানেন। আপনি আরও জানেন কিভাবে স্বজ্ঞাতভাবে বের করতে হয় যে একটি অ্যালগরিদমের জটিলতা হল O(1), O(log(n)), O(n), O(n^2) ইত্যাদি। আপনি o, O, ω, Ω এবং Θ চিহ্নগুলি জানেন এবং সবচেয়ে খারাপ-কেস বিশ্লেষণের অর্থ কী। আপনি যদি এতদূর এসে থাকেন, এই টিউটোরিয়ালটি ইতিমধ্যেই তার উদ্দেশ্য পূরণ করেছে।

এই চূড়ান্ত বিভাগ ঐচ্ছিক. এটি একটু বেশি জড়িত, তাই আপনি যদি এটি দ্বারা অভিভূত বোধ করেন তবে নির্দ্বিধায় এটি এড়িয়ে যান। এর জন্য আপনাকে ফোকাস করতে হবে এবং অনুশীলনের মাধ্যমে কিছু মুহূর্ত কাজ করতে হবে। যাইহোক, এটি আপনাকে অ্যালগরিদম জটিলতা বিশ্লেষণে একটি খুব দরকারী পদ্ধতি প্রদান করবে যা খুব শক্তিশালী হতে পারে, তাই এটি অবশ্যই বোঝার যোগ্য।

আমরা উপরে একটি বাছাই বাস্তবায়নের দিকে তাকিয়েছি একটি নির্বাচন সাজানোর নামে। আমরা উল্লেখ করেছি যে নির্বাচন বাছাই সর্বোত্তম নয়। একটি *সর্বোত্তম অ্যালগরিদম* হল এমন একটি অ্যালগরিদম যা সম্ভাব্য সর্বোত্তম উপায়ে একটি সমস্যার সমাধান করে, যার অর্থ এর জন্য আরও ভাল অ্যালগরিদম নেই। এর মানে হল যে সমস্যা সমাধানের জন্য অন্যান্য সমস্ত অ্যালগরিদমের সেই সর্বোত্তম অ্যালগরিদমের চেয়ে খারাপ বা সমান জটিলতা রয়েছে। একটি সমস্যার জন্য অনেকগুলি সর্বোত্তম অ্যালগরিদম থাকতে পারে যে সমস্ত একই জটিলতা ভাগ করে। বাছাই সমস্যাটি বিভিন্ন উপায়ে সর্বোত্তমভাবে সমাধান করা যেতে পারে। দ্রুত সাজানোর জন্য আমরা বাইনারি অনুসন্ধানের মতো একই ধারণা ব্যবহার করতে পারি। এই বাছাই পদ্ধতিকে *মার্জসর্ট* বলা হয় ।

একটি মার্জসর্ট সম্পাদন করতে, আমাদের প্রথমে একটি সহায়ক ফাংশন তৈরি করতে হবে যা আমরা তারপরে প্রকৃত সাজানোর জন্য ব্যবহার করব। আমরা একটি ফাংশন তৈরি করব `merge`যা দুটি অ্যারে নেয় যা উভয়ই ইতিমধ্যে সাজানো এবং একটি বড় সাজানো অ্যারেতে একত্রিত করে। এটি সহজেই করা হয়:

```
01.<font style="vertical-align: inherit;"><font style="vertical-align:
inherit;">            def মার্জ (A, B):</font></font><font></font><font style="vertical-
align: inherit;"><font style="vertical-align: inherit;">

02.যদি খালি (A):</font></font><font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">

03.ফেরত বি</font></font><font></font><font style="vertical-align: inherit;"><font
style="vertical-align: inherit;">
```

04.যদি খালি থাকে (B):

05.ফেরত A

06.যদি A[0] < B[0]:<font style="vertical-align:
inherit;">

07.রিটার্ন কনক্যাট(A[0], মার্জ(A[1...A_n], B))

08.অন্য:

09.রিটার্ন কনক্যাট(B[0], মার্জ(A, B[1...B_n]))

কনক্যাট ফাংশনটি একটি আইটেম, "হেড" এবং একটি অ্যারে, "টেইল" নেয় এবং একটি নতুন অ্যারে তৈরি করে এবং ফেরত দেয় যাতে প্রদত্ত "হেড" আইটেমটি নতুন অ্যারের প্রথম জিনিস এবং প্রদত্ত "টেইল" ধারণ করে " অ্যারের বাকি উপাদান হিসাবে আইটেম। উদাহরণস্বরূপ, concat(3, [4, 5, 6]) ফেরত দেয় [3, 4, 5, 6]। আমরা যথাক্রমে A এবং B এর আকার বোঝাতে A_n এবং B_n ব্যবহার করি।

ব্যায়াম 8

যাচাই করুন যে উপরের ফাংশনটি আসলে একটি মার্জ করে। forপুনরাবৃত্তি ব্যবহার করার পরিবর্তে এটিকে আপনার প্রিয় প্রোগ্রামিং ভাষায় পুনরাবৃত্ত উপায়ে (লুপ ব্যবহার করে) পুনরায় লিখুন।

এই অ্যালগরিদম বিশ্লেষণ করলে দেখা যায় যে এটির একটি চলমান সময় রয়েছে Θ(n), যেখানে n হল ফলাফলের অ্যারের দৈর্ঘ্য (n = A_n + B_n)।

ব্যায়াম 9

যাচাই করুন যে চলমান সময় mergeহল Θ(n)।

এই ফাংশনটি ব্যবহার করে আমরা একটি ভাল সাজানোর অ্যালগরিদম তৈরি করতে পারি। ধারণাটি নিম্নরূপ: আমরা অ্যারেটিকে দুটি অংশে বিভক্ত করি। আমরা দুটি অংশের প্রতিটিকে পুনরাবৃত্তভাবে সাজাই, তারপর আমরা দুটি সাজানো অ্যারেকে একটি বড় অ্যারেতে মার্জ করি। সিউডোকোডে:

1.
def mergeSort(A, n):

2.যদি n = 1:

3.ফেরত A # এটি ইতিমধ্যে সাজানো আছে

4.মধ্যম = মেঝে (n / 2)

5.বামহাফ = A[1... মধ্য]

6.ডানহাফ = A [(মধ্য + 1) ... n]

7.রিটার্ন মার্জ(mergeSort(leftHalf, Middle), mergeSort(rightHalf, n - Middle))

আমরা আগে যা করেছি তার থেকে এই ফাংশনটি বোঝা কঠিন, তাই নিম্নলিখিত অনুশীলনটি আপনার কয়েক মিনিট সময় নিতে পারে।

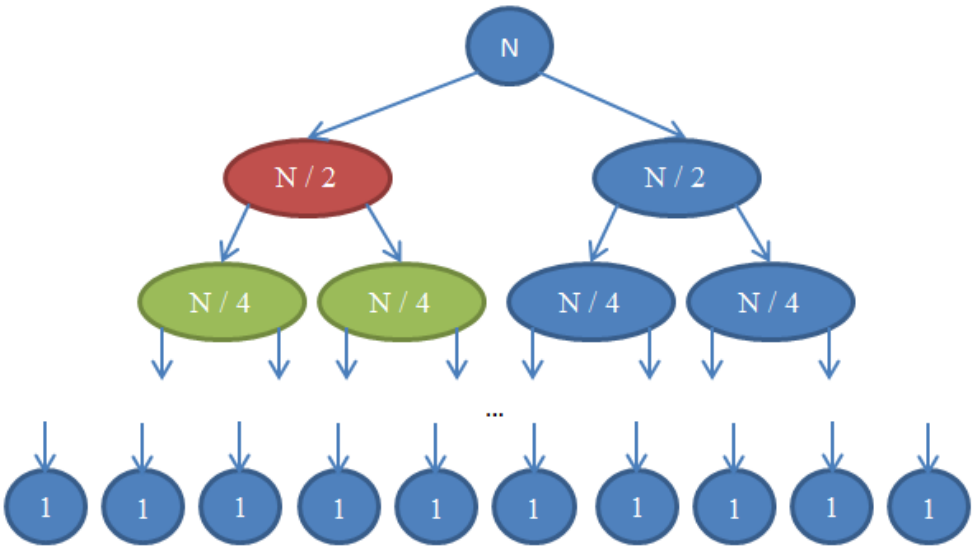
ব্যায়াম 10

এর সঠিকতা যাচাই করুন mergeSort। অর্থাৎ, mergeSortউপরে বর্ণিত অ্যারেটি আসলে সঠিকভাবে সাজায় কিনা তা পরীক্ষা করে দেখুন। এটি কেন কাজ করে তা বুঝতে আপনার সমস্যা হলে, একটি ছোট উদাহরণ অ্যারে দিয়ে চেষ্টা করুন এবং "হাত দিয়ে" চালান। হাত দিয়ে এই ফাংশনটি চালানোর সময়, নিশ্চিত করুন যে আপনি প্রায় মাঝখানে অ্যারেটি কেটে ফেললে বামহাফ এবং ডানহাফ যা পাবেন; অ্যারেতে বিজোড় সংখ্যক উপাদান থাকলে এটি ঠিক মাঝখানে থাকতে হবে না (floorউপরে এর জন্য ব্যবহার করা হয়েছে)।

একটি চূড়ান্ত উদাহরণ হিসাবে, এর জটিলতা বিশ্লেষণ করা যাক mergeSort। এর প্রতিটি ধাপে mergeSort, আমরা অ্যারেটিকে সমান আকারের দুটি অংশে বিভক্ত করছি, একইভাবে binarySearch। যাইহোক, এই ক্ষেত্রে, আমরা মৃত্যুদন্ড জুড়ে উভয় অর্ধেক বজায় রাখি। তারপরে আমরা প্রতিটি অর্ধেক পুনরাবৃত্তভাবে অ্যালগরিদম প্রয়োগ করি। পুনরাবৃত্তির পর, আমরা mergeফলাফলের উপর অপারেশন প্রয়োগ করি যা Θ(n) সময় নেয়।

সুতরাং, আমরা মূল অ্যারেটিকে প্রতিটি n/2 আকারের দুটি অ্যারেতে বিভক্ত করি। তারপরে আমরা সেই অ্যারেগুলিকে একত্রিত করি, একটি অপারেশন যা *n* উপাদানগুলিকে একত্রিত করে এবং এইভাবে Θ(n) সময় নেয়।

এই পুনরাবৃত্তি বোঝার জন্য **চিত্র 7** দেখুন ।



দেখা যাক এখানে কি হচ্ছে. প্রতিটি বৃত্ত ফাংশন একটি কল প্রতিনিধিত্ব করে `mergeSort`. বৃত্তে লেখা সংখ্যাটি সাজানো হচ্ছে অ্যারের আকার নির্দেশ করে। উপরের নীল বৃত্তটি হল মূল কল , যেখানে আমরা *n*`mergeSort` আকারের একটি অ্যারে সাজাতে পারি । তীরগুলি ফাংশনের মধ্যে করা পুনরাবৃত্তিমূলক কলগুলি নির্দেশ করে। দুটি অ্যারেতে দুটি কল করার জন্য আসল কল , প্রতিটির আকার n / 2। এটি উপরের দিকে দুটি তীর দ্বারা নির্দেশিত হয়। পালাক্রমে , এই কলগুলির প্রত্যেকটি n/4 আকারের দুটি অ্যারেতে নিজস্ব দুটি কল করে , এবং যতক্ষণ না আমরা আকার 1 এর অ্যারেতে পৌঁছাই / এবং দেখতে একটি গাছের মতো (*মূল* শীর্ষে এবং *পাতাগুলি* নীচে, তাই বাস্তবে এটি একটি বিপরীত গাছের মতো দেখায়)।`mergeSortmergeSortmergeSort`

লক্ষ্য করুন যে উপরের চিত্রের প্রতিটি সারিতে, মোট উপাদানের সংখ্যা n। এটি দেখতে, প্রতিটি সারি আলাদাভাবে দেখুন। প্রথম সারিতে *n*`mergeSort` আকারের অ্যারের সাথে শুধুমাত্র একটি কল আছে , তাই উপাদানের মোট সংখ্যা হল *n* । দ্বিতীয় সারিতে n / 2 আকারের প্রতিটিতে দুটি কল রয়েছে। কিন্তু n / 2 + n / 2 = n এবং তাই আবার এই সারিতে মোট উপাদানের সংখ্যা *n* । তৃতীয় সারিতে, আমাদের কাছে 4টি কল রয়েছে যার প্রতিটি একটি n/4-আকারের অ্যারেতে প্রয়োগ করা হয়েছে, যা n/4 + n/4 + n/4 + n/4 = 4n/4 এর সমান উপাদানের মোট সংখ্যা দেয়। = n. তাই আবার আমরা *n* উপাদান পেতে. এখন লক্ষ্য করুন যে এই ডায়াগ্রামের প্রতিটি সারিতে কলকারীকে কলিজ দ্বারা ফিরে আসা উপাদানগুলির উপর একটি অপারেশন করতে হবে। উদাহরণস্বরূপ, লাল রঙ দিয়ে নির্দেশিত বৃত্তটিকে n / 2 উপাদানগুলি সাজাতে হবে। এটি করার জন্য, এটি n / 2-আকারের অ্যারেটিকে দুটি n / 4-আকারের অ্যারেতে বিভক্ত করে, সেগুলিকে সাজানোর জন্য পুনরাবৃত্তিমূলকভাবে কল করে (এই কলগুলি হল সবুজ রঙ দিয়ে নির্দেশিত বৃত্ত), তারপর সেগুলিকে একত্রিত করে। এই মার্জ অপারেশনের জন্য n/2 উপাদান একত্রিত করতে হবে। আমাদের গাছের প্রতিটি সারিতে, একত্রিত উপাদানের মোট সংখ্যা n। যে সারিতে আমরা এইমাত্র অন্বেষণ করেছি, আমাদের ফাংশনটি n/2 উপাদানগুলিকে একত্রিত করে এবং এর ডানদিকের ফাংশনটি (যেটি নীল রঙে) তার নিজস্ব n/2 উপাদানগুলিকে একত্রিত করতে হবে। এটি মোট n উপাদান দেয় যা আমরা যে সারিটির দিকে তাকাচ্ছি তার জন্য মার্জ করা দরকার।`mergeSortmergemergeSort`

এই যুক্তি দ্বারা, প্রতিটি সারির জটিলতা হল Θ(n)। আমরা জানি যে এই ডায়াগ্রামে সারির সংখ্যা, যাকে পুনরাবৃত্তি গাছের *গভীরতাও* বলা হয় , লগ(n) হবে। বাইনারি অনুসন্ধানের জটিলতা বিশ্লেষণ করার সময় আমরা যেটি ব্যবহার করেছি তার মতোই এর যুক্তি। আমাদের log(n) সারি রয়েছে এবং তাদের প্রতিটি হল Θ(n), তাই এর জটিলতা `mergeSort`হল Θ(n * log(n))। এটি Θ(n^2) এর চেয়ে অনেক ভাল যা নির্বাচনের সাজানো আমাদের দিয়েছে (মনে রাখবেন যে log(n) n এর চেয়ে অনেক ছোট, এবং তাই n * log(n) n * n = n^2 এর চেয়ে অনেক ছোট)। যদি এটি

আপনার কাছে জটিল মনে হয়, তবে চিন্তা করবেন না: আপনি প্রথমবার এটি দেখতে সহজ নয়। আপনি আপনার প্রিয় প্রোগ্রামিং ভাষায় মার্জট প্রয়োগ করার পরে এবং এটি কাজ করে তা যাচাই করার পরে এই বিভাগে পুনরায় যান এবং এখানে আর্গুমেন্টগুলি পুনরায় পড়ুন।

আপনি এই শেষ উদাহরণে যেমন দেখেছেন, জটিলতা বিশ্লেষণ আমাদেরকে অ্যালগরিদম তুলনা করার অনুমতি দেয় কোনটি ভাল তা দেখতে। এই পরিস্থিতিতে, আমরা এখন মোটামুটি নিশ্চিত হতে পারি যে মার্জ সর্ট বৃহৎ অ্যারেগুলির জন্য নির্বাচনের সাজানোর কাজকে ছাড়িয়ে যাবে। এই উপসংহার টানা কঠিন হবে যদি আমাদের তৈরি করা অ্যালগরিদম বিশ্লেষণের তাত্ত্বিক পটভূমি না থাকে। অনুশীলনে, প্রকৃতপক্ষে চলমান সময়ের Θ(n * log(n)) বাছাই করার অ্যালগরিদম ব্যবহার করা হয়। উদাহরণস্বরূপ, লিনাক্স কার্নেল একটি সাজানোর অ্যালগরিদম ব্যবহার করে যাকে বলা হয় heapsort , যার চলমান সময়টি mergesort-এর মতোই রয়েছে যা আমরা এখানে অন্বেষণ করেছি, যথা Θ(n log(n)) এবং তাই সর্বোত্তম। লক্ষ্য করুন যে আমরা প্রমাণ করিনি যে এই সাজানোর অ্যালগরিদমগুলি সর্বোত্তম। এটি করার জন্য একটি সামান্য বেশি জড়িত গাণিতিক যুক্তি প্রয়োজন, তবে নিশ্চিত থাকুন যে তারা জটিলতার দৃষ্টিকোণ থেকে আরও ভাল পেতে পারে না।

এই টিউটোরিয়ালটি পড়া শেষ করার পরে, অ্যালগরিদম জটিলতা বিশ্লেষণের জন্য আপনি যে অন্তর্দৃষ্টি তৈরি করেছেন তা আপনাকে দ্রুত প্রোগ্রাম ডিজাইন করতে এবং আপনার অপ্টিমাইজেশান প্রচেষ্টাগুলিকে গুরুত্বপূর্ণ বিষয়গুলির পরিবর্তে গুরুত্বপূর্ণ বিষয়গুলিতে ফোকাস করতে সক্ষম হবে, যা আপনাকে আরও উত্পাদনশীলভাবে কাজ করতে দেয়। . এছাড়াও, এই নিবন্ধে তৈরি করা গাণিতিক ভাষা এবং স্বরলিপি যেমন big-O স্বরলিপি অন্যান্য সফ্টওয়্যার ইঞ্জিনিয়ারদের সাথে যোগাযোগ করতে সহায়ক যখন আপনি অ্যালগরিদমের চলমান সময় সম্পর্কে তর্ক করতে চান, তাই আশা করি আপনি এটি করতে সক্ষম হবেন আপনার নতুন অর্জিত জ্ঞান।

সম্পর্কিত

আমি বর্তমানে এথেন্স বিশ্ববিদ্যালয়ে ক্রিপ্টোগ্রাফি পিএইচডি প্রার্থী । আমি যখন এই নিবন্ধটি লিখেছিলাম তখন আমি ন্যাশনাল টেকনিক্যাল ইউনিভার্সিটি অফ এথেন্সে স্নাতক ইলেকট্রিক্যাল এবং কম্পিউটার ইঞ্জিনিয়ারিংয়ে স্নাতক ছিলাম এবং সফটওয়্যারে মাস্টার্স করছিলাম এবং ইনফরম্যাটিক্সে গ্রীক প্রতিযোগিতার একজন প্রশিক্ষক ছিলাম । ইন্ডাস্ট্রি অনুসারে আমি ইঞ্জিনিয়ারিং টিমের একজন সদস্য হিসাবে কাজ করেছি যেটি Google এবং Twitter এর নিরাপত্তা দলে, শিল্পীদের জন্য একটি সামাজিক নেটওয়ার্ক deviantART তৈরি করেছে এবং দুটি স্টার্ট-আপ, Zino এবং Kamibu যেখানে আমরা সামাজিক নেটওয়ার্কিং এবং ভিডিও গেম করেছি যথাক্রমে উন্নয়ন। আপনি যদি এটি উপভোগ করেন তবে আমাকে Twitter বা GitHub- এ অনুসরণ করুন , অথবা আপনি যদি যোগাযোগ করতে চান তবে আমাকে মেইল

করুন । অনেক তরুণ প্রোগ্রামারদের ইংরেজি ভাষার ভালো জ্ঞান নেই। আপনি যদি এই নিবন্ধটি আপনার নিজের মাতৃভাষায় অনুবাদ করতে চান তাহলে আমাকে ই-মেইল করুন যাতে আরও বেশি লোক এটি পড়তে পারে।

পড়ার জন্য ধন্যবাদ। আমি এই নিবন্ধটি লেখার জন্য অর্থ প্রদান করিনি, তাই যদি আপনি এটি পছন্দ করেন, হ্যালো বলার জন্য আমাকে একটি ই-মেইল পাঠান । আমি সারা বিশ্বের স্থানের ছবি প্রাপ্তি উপভোগ করি, তাই নির্দ্বিধায় আপনার শহরে নিজের একটি ছবি সংযুক্ত করুন!

তথ্যসূত্র

1. কোরমেন, লিজারসন, রিভেস্ট, স্টেইন। অ্যালগরিদমের ভূমিকা , এমআইটি প্রেস।
2. দাশগুপ্ত, পাপদিমিত্রিউ, ভাজিরানি। অ্যালগরিদম , ম্যাকগ্রা-হিল প্রেস।
3. ফোটাকিস। এথেন্সের ন্যাশনাল টেকনিক্যাল ইউনিভার্সিটিতে ডিসক্রিট ম্যাথমেটিক্সের কোর্স ।
4. ফোটাকিস। অ্যাথেন্সের ন্যাশনাল টেকনিক্যাল ইউনিভার্সিটিতে অ্যালগরিদম এবং জটিলতার কোর্স ।