ساختمان داده و الگوریتم در

جاوااسکریپت

Sammie Bae

ویرایش نخست

سید منصور عمرانی

انتشارات پندار پارس

سرشناسه	:	بائه، سامی Bae, Sammie
عنوان و نام پدیدآور	:	ساختمان داده و الگوریتم در جاوا اسکریپت/] تالیف سمی بی[؛] ترجمه [سیدمنصور عمرانی.
مشخصات نشر	:	تهران : پندار پارس، ۱۴۰۱.
مشخصات ظاهری	:	۳۰۸ص.
شابک	:	978-622-7785-06-7:
وضعیت فهرست نویسی	:	فیپا
یادداشت	:	عنوان اصلی: JavaScript Data Structures and Algorithms : An Introduction to Understanding and Implementing Core Data Structure and Algorithm Fundamentals, 2019.
موضوع	:	برنامه‌نویسی Computer programming
شناسه افزوده	:	عمرانی، سیدمنصور، ۱۳۵۶ -، مترجم
رده بندی کنگره	:	QA۷۶/۶
رده بندی دیویی	:	۷۶/۰۰۶
شماره کتابشناسی ملی	:	۸۸۱۲۲۶۹
اطلاعات رکورد کتابشناسی	:	فیپا

انتشارات پندارپارس

دفتر فروش: انقلاب، ابتدای کارگرجنوبی، کوی رشتچی، شماره ۱۴، واحد ۱۶ **www.pendarepars.com**

تلفن: ۶۶۵۷۲۳۳۵ – تلفکس: ۶۶۹۲۶۵۷۸همراه: ۰۹۱۲۲۴۵۲۳۴۸ **info@pendarepars.com**

نام کتاب : **ساختمان داده و الگوریتم در جاوااسکریپت**

ناشر : **انتشارات پندار پارس**

تالیف : **سمی بی**

ترجمه : **سید منصور عمرانی**

چاپ نخست : **اردیبهشت ۱۴۰۱**

شمارگان : **۱۰۰ نسخه دیجیتال**

طرح جلد : **رامین شکرالهی**

چاپ، صحافی : **روز** **چاپ با کاغذ یارانه‌ای تخصیصی از سوی وزارت ارشاد**

قیمت : **۱۶۵.۰۰۰ تومان** **شابک : ۷-۰۶-۷۷۸۵-۶۲۲-۹۷۸**

فهرست

فصل ۱. نماد O بزرگ (Big-O) ۱
اصول اولیه و مبانی نماد Big-O ۱
مثال‌های رایج ۲
قوانین نماد Big-O ۳
قانون ضریب: ثوابت را حذف کنید ۴
قانون جمع: O ها را با هم جمع کنید ۵
قانون ضرب: O ها را در هم ضرب کنید ۶
قانون چند جمله‌ای: O توان k ۶
خلاصه ۷
تمرین ۷
فصل ۲. نکته‌های مهم زبان جاوااسکریپت ۹
حوزه‌ی دید ۹
تعریف متغیر به صورت سراسری: حوزه‌ی دید سراسری ۹
متغیرهای var: حوزه‌ی دید تابعی ۹
متغیرهای let: حوزه‌ی دید بلاکی ۱۲
مقایسه‌ی برابری ۱۲
انواع متغیر ۱۲
بررسی درستی (truthy) و نادرستی (falsey) ۱۳
عملگر === و == ۱۴
مقایسه‌ی اشیا ۱۴
خلاصه ۱۶
فصل ۳. اعداد در جاوااسکریپت ۱۷
سیستم عددی ۱۷
شی Number ۱۹
تقسیم صحیح و رُند کردن ۱۹
Number.EPSILON ۱۹
ماگزیمم ۲۰
می‌نیمم ۲۰
بینهایت یا infinity ۲۱
خلاصه‌ی بزرگی و کوچکی مقادیر ثابت جاوااسکریپت ۲۱
الگوریتم‌های عددی ۲۱
آزمایش اول بودن ۲۱
به دست آوردن فاکتورهای اول یک عدد ۲۳
تولید اعداد تصادفی ۲۳
تمرین ۲۴
خلاصه ۲۷

فصل ۴. رشته‌ها در جاوااسکریپت ۲۹

کلاس پایه‌ای String در جاوااسکریپت ۲۹

مقایسه‌ی رشته ۲۹

جستجو در رشته ۳۰

خُرد کردن رشته ۳۱

جایگزین کردن رشته ۳۱

عبارت‌های با قاعده ۳۲

مبانی عبارت با قاعده ۳۲

چند عبارت با قاعده‌ی رایج و مفید ۳۳

Query String ۳۴

انکُد کردن رشته‌ها ۳۵

انکُدینگ Base64 ۳۵

کوتاه کردن رشته ۳۵

رمزنگاری ۳۷

الگوریتم رمزنگاری RSA ۳۹

خلاصه ۴۳

فصل ۵. آرایه‌ها ۴۵

معرفی ۴۵

درج ۴۵

حذف ۴۵

دستیابی ۴۶

تکرار یا Iteration ۴۶

دستور for (; ;) ۴۶

دستور while ۴۷

دستور for (in) ۴۷

دستور for (of) ۴۸

متد forEach() ۴۸

توابع کمکی ۴۹

.slice(beginIndex, endIndex) ۴۹

تابع splice(startIndex, removeCount, element1, element2, ...) ۴۹

concat() ۵۱

خصوصیت length ۵۱

عملگر Spread ۵۲

تعریف تابعی با تعداد پارامتر متغیر ۵۲

پاس دادن تعداد آرگومان متغیر به تابع ۵۲

تمرین ۵۳

متدهای تابعی کار با آرایه ۵۹

map() ۵۹

filter() ۵۹

reduce() ۵۹

آرایه‌های چند بُعدی ۶۰

تمرین ۶۲
خلاصه ۷۱
فصل ۶. اشیاء ۷۳
خصوصیت‌های اشیاء ۷۳
وراثت پروتوتایپی ۷۳
سازنده و متغیرها ۷۴
خلاصه ۷۵
تمرین ۷۵
فصل ۷. مدیریت حافظه در جاوااسکریپت ۷۷
نشتی حافظه ۷۷
ارجاع به اشیا ۷۷
نشتی DOM ۷۸
شی سراسری window ۷۹
محدود کردن ارجاع به اشیا ۷۹
عملگر delete ۸۰
خلاصه ۸۰
تمرین ۸۰
فصل ۸. الگوریتم‌های بازگشتی ۸۵
معرفی بازگشت ۸۵
قوانین بازگشت ۸۵
حالت پایه ۸۶
روش تقسیم و غلبه ۸۶
مثال کلاسیک: سری فیبوناچی ۸۶
حل مساله به روش تکرار ۸۷
حل مساله به روش بازگشتی ۸۷
حل مساله به روش tail recursion ۸۸
مثلث پاسکال ۸۹
محاسبه‌ی پیچیدگی زمانی Big-O برای توابع بازگشتی ۹۰
روابط تکرار دوباره (Recurrence relations) ۹۰
تئوری مستر ۹۱
حافظه‌ی پشته‌ی فراخوانی بازگشتی ۹۲
خلاصه ۹۳
تمرین ۹۴
فصل ۹. مجموعه‌ها ۹۹
معرفی مجموعه‌ها ۹۹
عملیات کار با مجموعه ۹۹
درج (Insertion) ۱۰۰

حذف (Deletion) ۱۰۰
شامل بودن (Contains) ۱۰۰
دیگر توابع مفید کار با مجموعه ۱۰۰
اشتراک ۱۰۱
چک کردن مجموعه‌ی پدر بودن ۱۰۱
اجتماع ۱۰۱
افتراق ۱۰۲
خلاصه ۱۰۲
تمرین ۱۰۳
فصل ۱۰. جستجو و مرتب‌سازی ۱۰۵
جستجو ۱۰۵
جستجوی خطی ۱۰۵
جستجوی دودویی ۱۰۶
مرتب‌سازی ۱۰۸
مرتب‌سازی حبابی ۱۰۸
مرتب‌سازی انتخابی ۱۱۰
مرتب‌سازی درجی ۱۱۱
مرتب‌سازی سریع ۱۱۳
انتخاب سریع ۱۱۵
مرتب‌سازی ادغامی ۱۱۶
مرتب‌سازی شمارشی ۱۱۸
تابع درون ساخت مرتب‌سازی آرایه‌ها در جاوااسکریپت ۱۱۹
خلاصه ۱۲۰
تمرین ۱۲۱
فصل ۱۱. جدول هَش ۱۲۷
معرفی جدول هَش ۱۲۷
تکنیک‌های هَش کردن ۱۲۸
استفاده از اعداد اول ۱۲۸
کاوش ۱۳۰
کاوش خطی ۱۳۰
کاوش درجه دو ۱۳۰
هَش کردن دوباره/دو بار هَش کردن ۱۳۱
پیاده‌سازی جدول هَش ۱۳۱
استفاده از تکنیک کاوش خطی ۱۳۲
استفاده از تکنیک کاوش درجه دو ۱۳۳
استفاده از تکنیک دو بار هَش کردن و کاوش خطی ۱۳۴
خلاصه ۱۳۵
فصل ۱۲. پشته و صف ۱۳۷
پشته ۱۳۷
عمل Peek یا نگاه کردن ۱۳۸

درج ۱۳۸
حذف ۱۳۹
دستیابی ۱۳۹
جستجو ۱۴۰
صف ۱۴۰
عمل Peek یا نگاه کردن ۱۴۲
درج ۱۴۲
دستیابی ۱۴۳
جستجو ۱۴۳
خلاصه ۱۴۴
تمرین ۱۴۴
فصل ۱۳. لیست پیوندی ۱۴۹
لیست پیوندی یک طرفه ۱۴۹
درج ۱۵۰
حذف ۱۵۰
حذف سر لیست ۱۵۲
جستجو ۱۵۲
لیست پیوندی دو طرفه ۱۵۳
درج در سر لیست ۱۵۴
درج در ته لیست ۱۵۴
حذف گره سر لیست ۱۵۵
حذف گره ته لیست ۱۵۵
جستجو ۱۵۶
خلاصه ۱۵۷
تمرین ۱۵۸
فصل ۱۴. حافظه‌ی کَش ۱۶۱
درک مفهوم کَش ۱۶۱
ساختار کلی کَش ۱۶۲
تکنیک کمترین دفعات استفاده یا LFU ۱۶۲
پیاده‌سازی کَش LFU ۱۶۳
پیاده‌سازی متد ()set و ()get ۱۶۴
تکنیک کمترین دسترسی اخیر یا LRU ۱۶۷
خلاصه ۱۷۰
فصل ۱۵. درخت ۱۷۱
ساختار کلی درخت ۱۷۱
درخت دودویی ۱۷۲
پیمایش درخت ۱۷۲
پیمایش Pre-Order ۱۷۲
پیمایش In_order ۱۷۴
پیمایش Post-Order ۱۷۶

پیمایش Level-Order ۱۷۷

خلاصه‌ی روش‌های پیمایش درخت ۱۷۸

درخت جستجوی دودویی ۱۷۹

درج ۱۷۹

حذف ۱۸۱

جستجو ۱۸۲

درخت AVL ۱۸۳

دوران تکی ۱۸۴

دوران به چپ ۱۸۴

دوران مضاعف ۱۸۶

دوران به راست، سپس به چپ ۱۸۶

دوران به چپ، سپس به راست ۱۸۷

متوازن کردن درخت ۱۸۸

درج ۱۸۹

حذف ۱۹۰

سر هم کردن همه‌ی اجزا ۱۹۱

خلاصه ۱۹۲

تمرین ۱۹۳

فصل ۱۶. هیپ ۲۰۱

هیپ چیست؟ ۲۰۱

Max-Heap ۲۰۲

Min-Heap ۲۰۲

ساختار اندیس آرایه‌ی هیپ ۲۰۳

نفوذ: بالا و پایین رفتن حبابی ۲۰۴

پیاده‌سازی عملیات percolation ۲۰۶

مثالی از Max-Heap ۲۰۷

درج و حذف ۲۰۹

پیاده‌سازی Max-Heap ۲۱۰

الگوریتم مرتب‌سازی هیپ (heap sort) ۲۱۲

مرتب‌سازی صعودی با min-heap ۲۱۲

مرتب‌سازی نزولی با max-heap ۲۱۴

خلاصه ۲۱۶

تمرین ۲۱۶

فصل ۱۷. گراف ۲۲۱

مبانی گراف ۲۲۱

گراف غیرجهت‌دار ۲۲۴

افزودن یال و گره ۲۲۵

حذف گره‌ها و یال‌ها ۲۲۶

گراف جهت‌دار ۲۲۸

پیمایش گراف ۲۳۱
جستجوی اول سطح (Breadth-First Search) ۲۳۱
جستجوی اول عمق (Depth-First Search) ۲۳۵
گراف وزن‌دار و کوتاهترین مسیر ۲۳۹
گراف وزن‌دار ۲۳۹
الگوریتم دیجکسترا: پیدا کردن کوتاهترین مسیر ۲۴۰
مرتب‌سازی توپولوژیکی ۲۴۴
خلاصه ۲۴۶
فصل ۱۸. مباحث پیشرفته‌ی کار با رشته‌ها ۲۴۹
درخت پیشوند (Prefix Tree یا Trie) ۲۴۹
الگوریتم جستجوی رشته‌ی بویر-مور ۲۵۲
الگوریتم جستجوی رشته‌ی ناث-موریس-پرت ۲۵۶
جستجوی رابین-کارپ ۲۵۹
اثر انگشت رابین ۲۶۰
برنامه‌های دنیای واقعی ۲۶۳
خلاصه ۲۶۳
فصل ۱۹. برنامه‌نویسی پویا ۲۶۵
انگیزه‌ی برنامه‌نویسی پویا ۲۶۵
قوانین برنامه‌نویسی پویا ۲۶۷
همپوشانی زیر مساله‌ها ۲۶۷
زیر ساختار بهینه یا مطلوب ۲۶۷
مثال: راه‌های پوشش دادن مراحل ۲۶۷
مثال‌های کلاسیکی از برنامه‌نویسی پویا ۲۶۹
مساله‌ی کوله پشتی ۲۶۹
زیر ساختار بهینه ۲۶۹
راه حل ناپخته ۲۶۹
رهیافت DP ۲۷۰
بزرگترین زیر دنباله‌ی مشترک ۲۷۱
روش ناپخته ۲۷۱
رهیافت DP ۲۷۳
تغییر سکه ۲۷۳
زیر ساختار بهینه ۲۷۴
رهیافت ناپخته ۲۷۴
همپوشانی زیر مساله‌ها ۲۷۴
رهیافت DP ۲۷۵
فاصله‌ی ویرایش (لونشتاین) ۲۷۶
زیر ساختار بهینه ۲۷۶
رهیافت ناپخته ۲۷۷
رهیافت DP ۲۷۸
خلاصه ۲۷۹

فصل ۲۰. عملیات بیتی ۲۸۱

عملگرهای بیتی ۲۸۱

عملگر AND یا & ۲۸۱

عملگر OR یا | ۲۸۲

یای انحصاری، عملگر XOR یا ^ ۲۸۲

عملگر NOT یا ~ ۲۸۳

عملگر Shift به چپ یا << ۲۸۳

عملگر Shift به راست یا >> ۲۸۴

عملگر Shift به راست با وارد کردن بیت 0 یا >>> ۲۸۵

پیاده‌سازی عملگرهای محاسباتی به صورت بیتی ۲۸۵

جمع ۲۸۵

تفریق ۲۸۶

ضرب ۲۸۷

تقسیم ۲۸۸

خلاصه ۲۹۰

این کتاب را تقدیم می‌کنم به دکتر حمید آذرهوش برای الهام بخشیدن به من در مطالعاتم و همچنین به مادرم، مین کیونگ سئو، بخاطر مهربانی و حمایتش.

سمی بی

درباره‌ی نویسنده

سمی بی یکی از مهندسین داده‌ی شرکت Yelp در سان فرانسیسکو است و پیش از این در تیم مهندسی سکوی دیتای شرکت NVIDIA کار می‌کرد. او در شرکت SMART Technologies (که توسط شرکت Foxconn خریداری شد) به عنوان کارآموز فعالیت داشت و برای ارتباط یک برنامه‌ی وب و درایورهای بورد الکترونیکی از طریق پورت سریال با استفاده از Node.js، API می‌نوشت. در همین حین به جاوااسکریپت بسیار علاقمند شد. با وجود ارتباط بسیار زیاد جاوااسکریپت با صنعت پیشرفته‌ی مهندسی نرم‌افزار در حال حاضر هیچ کتابی به جز کتاب فعلی وجود ندارد که ساختمان داده و الگوریتم را به این زبان آموزش بدهد. سَمی می‌داند چقدر این مفاهیم علوم کامپیوتر سخت و پیچیده است. هدف او در این کتاب این است که توضیحی واضح و خلاصه برای این مفاهیم فراهم کند.

درباره‌ی ویراستار فنی

فیل نش یکی از برنامه‌نویسان Twilio است که در لندن و سراسر دنیا به جوامع برنامه‌نویسی خدمت می‌کند. او یک برنامه‌نویس Ruby، Javascript و Swift بوده و همچنین یکی از برنامه‌نویسان حرفه‌ای گوگل، یک سخنران و وب‌لاگ‌نویس است. می‌توان او را در کنفرانس‌ها، نشست‌ها و همایش‌های مختلف در حال ارائه‌ی محتوا در خصوص فناوری‌های پیشرفته و API یا نوشتن کُدهای اُپن‌سورس پیدا کرد.

تقدیر و تشکر

فیل نش، از بازخورد ارزشمندت که به من کمک کرد محتوای کتاب را با توضیحاتی واضح‌تر و مثال‌هایی خلاصه‌تر و کوتاهتر بهبود بدهم ممنوم.

از تیم انتشارات Apress نیز به طور ویژه سپاسگزارم، به خصوص از جیمز مارکهام، ننسی چن، جید اسکارد و کریس نلسون. در نهایت نیز مایلم از استیو انگلین تشکر کنم که مرا به همکاری با انتشارات Apress دعوت کرد.

مقدمه

انگیزه‌ی اصلی من در نوشتن این کتاب فقدان منابع کافی در زمینه‌ی ساختمان داده و الگوریتم به زبان جاوااسکریپت بود. این مساله بسیار برایم تعجب داشت، زیرا این روزها بسیاری از موقعیت‌های شغلی برنامه‌نویسی مستلزم آشنایی با جاوااسکریپت است. جاوااسکریپت تنها زبانی است که با دانستن آن می‌توان در همه‌ی بخش‌های پشته‌ی برنامه‌نویسی یک پروژه شامل front-end، موبایل (به صورت بومی یا دورگه) و back-end کار کرد. اطلاع از ساختمان داده و طراحی الگوریتم برای برنامه‌نویسان جاوااسکریپت امری حیاتی است.

از این رو در این کتاب مفاهیم ساختمان داده و الگوریتم را به جای زبان‌های مرسومی چون جاوا یا C++ به زبان جاوااسکریپت بیان کرده‌ام. از آنجایی که در جاوااسکریپت بر خلاف زبان‌هایی مانند جاوا و C++ (که وراثت در آنها به صورت کلاسی است) از وراثت پروتوتایپی[1] پیروی می‌کند، نحوه‌ی بیان برخی از ساختمان‌های داده در این کتاب کمی فرق دارد. در روش مرسوم وراثت کلاسی یک کپی از کلاس پدر ایجاد شده و اشیا فرزند از ساختار آن پیروی می‌کنند. اما در وراثت پروتوتایپی از خود اشیا کپی شده و خصوصیت‌هایشان تغییر داده می‌شود.

در این کتاب نخست مبانی ریاضیات پایه مانند تحلیل O بزرگ یا Big-O بیان شده و سپس زیر بنای جاوااسکریپتی لازم مانند اشیا و نوع داده‌های پایه توضیح داده می‌شود. پس از آن ساختمان داده‌های پایه مانند لیست پیوندی، پشته، درخت، هیپ و گراف بیان می‌شود. در نهایت مفاهیم پیشرفته‌تری مانند الگوریتم‌های بهینه‌ی جستجوی رشته، الگوریتم‌های کَش کردن و برنامه‌نویسی پویا با جزئیات کامل توضیح داده می‌شود.

---

[1] Prototypical Inheritance Pattern

فصل ۱. نماد O بزرگ (Big-O)

O(1) مقدس است.

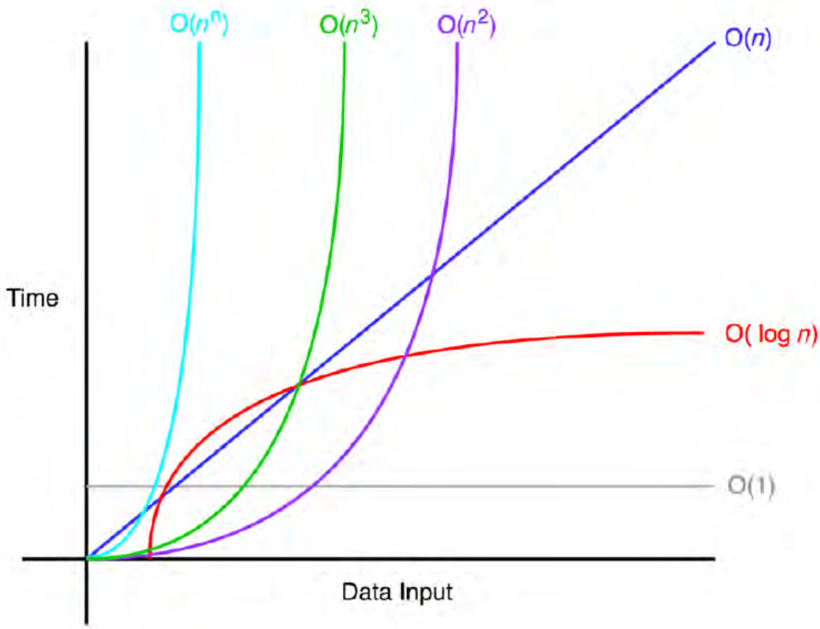
–حمید آذرهوش

پیش از یادگیری نحوه‌ی پیاده‌سازی الگوریتم‌ها نخست باید تحلیل کارایی آنها را بدانیم. در این فصل به نماد O بزرگ یا Big-O برای تحلیل پیچیدگی زمان و فضای الگوریتم‌ها می‌پردازیم. در انتهای این فصل نحوه‌ی تحلیل پیاده‌سازی یک الگوریتم را با در نظر داشتن زمان (زمان اجرا) و همچنین فضا (حافظه‌ی مصرف شده) یاد می‌گیریم.

اصول اولیه و مبانی نماد Big-O

نماد Big-O برای اندازه‌گیری پیچیدگی زمانی بدترین حالت یک الگوریتم به کار می‌رود. در نماد Big-O حرف n تعداد ورودی را نشان می‌دهد. سوالی که هنگام استفاده از Big-O مطرح می‌شود این است که «وقتی n به سمت بینهایت میل می‌کند چه اتفاقی برای الگوریتم می‌افتد؟»

وقتی الگوریتمی را پیاده‌سازی می‌کنید توجه به Big-O اهمیت زیادی دارد، زیرا به شما می‌گوید الگوریتم‌تان چقدر سریع بوده و کارایی‌اش چطور است. شکل ۱–۱ تعدادی از Big-O های مرسوم را نشان می‌دهد.



شکل ۱–۱. Big-O های رایج

در این قسمت این Big-O های رایج را با مثال‌های ساده توضیح می‌دهیم.

مثال‌های رایج

اگر پیچیدگی زمانی الگوریتمی O(1) باشد یعنی سرعت و کارایی آن با افزایش فضای ورودی تغییر نمی‌کند. در نتیجه O(1) به معنی زمان ثابت[1] است. یعنی هرچه حجم یا تعداد ورودی بیشتر و بزرگتر باشد زمان اجرا فرقی نکرده و ثابت است. نمونه‌ای از O(1) دسترسی به خانه‌های آرایه بر حسب اندیس است. مهم نیست طول یا حجم آرایه چقدر باشد. زمان دسترسی به خانه‌های آرایه همیشه ثابت است.

O(n) به معنی خطی بودن زمان[2] اجرای الگوریتم بوده و الگوریتم‌هایی را در بر می‌گیرد که در بدترین حالت اجرا باید n عمل انجام بدهند. در نتیجه زمان اجرا با افزایش فضا یا تعداد ورودی به طور خطی افزایش پیدا می‌کند. برای مثالی از O(n) می‌توان به چاپ اعداد صفر تا n - 1 مانند زیر اشاره کرد:

```
function exampleLinear(n) {
   for (var i = 0 ; i < n; i++ ) {
      console.log(i);
   }
}
```

به طور مشابه $O(n^2)$ و $O(n^3)$ به ترتیب به معنی زمان درجه دو (مربعی)[3] و درجه سه (مکعبی)[4] است. در زیر نمونه‌هایی از این دو پیچیدگی نشان داده شده است:

```
// quatratic time
function exampleQuadratic(n) {
  for (var i = 0 ; i < n; i++) {
     console.log(i);

     for (var j = i; j < n; j++) {
        console.log(j);
     }
  }
}
```

```
// cubic time
function exampleCubic(n) {
  for (var i = 0 ; i < n; i++) {
      console.log(i);
      for (var j = i; j < n; j++) {
          console.log(j);
          for (var k = j; j < n; j++) {
              console.log(k);
          }
      }
  }
}
```

در نهایت نمونه‌ای از الگوریتمی که پیچیدگی آن لگاریتمی یا $O(\log_n)$ باشد چاپ اعداد مربع بین 2 و n است. برای نمونه تابع exampleLogarithmic(100) در خروجی اعداد زیر را چاپ می‌کند:

2, 4, 8, 16, 32, 64

```
function exampleLogarithmic(n) {
   for (var i = 2 ; i <= n; i= i * 2 ) {
      console.log(i);
   }
}
```

[1] Constant time
[2] Linear time
[3] Quadratic time
[4] Cubic time

افت کارایی الگوریتم‌های با پیچیدگی لگاریتمی در تعداد ورودی بسیار بالا مانند میلیون آیتم ظاهر می‌شود. برای نمونه در مثال بالا اگر n برابر یک میلیون باشد با وجودی که حلقه‌ی تابع exampleLogarithmic() یک میلیون بار تکرار می‌شود اما در خروجی تنها ۱۹ عدد چاپ می‌شود زیرا $\log_2(1,000,000) = 19.9315686$ است.

قوانین نماد Big-O

فرض کنید پیچیدگی زمانی یک الگوریتم f(n) باشد. همان گونه که گفتیم n به معنی تعداد ورودی است. زمان مورد نیاز برای اجرای این الگوریتم با $f(n)_{time}$ و فضای مورد نیاز آن با $f(n)_{space}$ نشان داده می‌شود. هدف از تحلیل الگوریتم این است که بتوانیم میزان کارایی آن را با حساب کردن f(n) بفهمیم. با این حال محاسبه‌ی خود f(n) کار چالش‌برانگیزی است. از این رو بجای آن از نمادی به نام O بزرگ استفاده شده و قوانینی فراهم می‌شود تا برنامه‌نویسان با آن بتوانند f(n) را حساب کنند.

- **قانون ضریب[1]:** اگر f(n) برابر O(g(n)) باشد به ازای هر مقدار ثابت k بزرگتر از صفر kf(n) برابر O(g(n)) خواهد بود. قانون ضریب نخستین قانون نماد O است و ضرایبی را که ارتباطی با اندازه‌ی ورودی n ندارند حذف می‌کند (زیرا وقتی n به سمت بینهایت میل می‌کند از ضرایب می‌توان صرفنظر کرد).
- **قانون جمع[2]:** اگر f(n) برابر O(h(n)) و g(n) برابر O(p(n)) باشد f(n)+g(n) برابر O(h(n)+p(n)) خواهد بود. به بیان ساده اگر زمان اجرای الگوریتمی برابر جمع زمان اجرای دو الگوریتم مختلف باشد نماد O بزرگ آن نیز برابر جمع O بزرگ آن دو الگوریتم خواهد بود.
- **قانون ضرب[3]:** اگر پیچیدگی f(n) برابر O(h(n)) و پیچیدگی g(n) برابر O(p(n)) باشد در این صورت f(n)*g(n) برابر O(h(n)*p(n)) خواهد بود. به طور مشابه قانون ضرب بیان می‌کند وقتی زمان اجرا چند برابر شود نماد O بزرگ آن نیز چند برابر می‌شود.
- **قانون تعدی[4]:** اگر f(n) برابر O(g(n)) و g(n) برابر O(h(n)) باشد در این صورت f(n) برابر O(h(n)) خواهد بود. قانون تعدی یا تراگذری راه ساده‌ای برای بیان این مطلب است که پیچیدگی زمان اجرای الگوریتم‌های یکسان مانند هم است.
- **قانون چند جمله‌ای[5]:** اگر f(n) یک چند جمله‌ای درجه k باشد در این صورت پیچیدگی زمانی f(n) برابر $O(n^k)$ خواهد بود. همان گونه که مشاهده می‌شود قانون چند جمله‌ای بیان می‌کند پیچیدگی زمانی یک چندجمله‌ای نوع O همان درجه‌ی چند جمله‌ای است.
- **قانون لگاریتم توان[6]:** پیچیدگی زمانی log(nk) به ازای هر k بزرگتر از صفر برابر O(log(n)) است. طبق این قانون هنگام محاسبه‌ی O از ثوابت داخل توابع لگاریتمی صرفنظر می‌شود.

---

[1] Coefficient rule
[2] Sum rule
[3] Product rule
[4] Transitive rule
[5] Polynomial rule
[6] Log of a power rule

سه قانون اول و همچنین قانون چند جمله‌ای از بقیه مهم‌تر هستند، زیرا بیشترین کاربرد را دارند. در قسمت‌های پیش رو هر یک از این قوانین را توضیح می‌دهیم.

قانون ضریب: ثوابت را حذف کنید

بگذارید نخست قانون ضریب را بررسی کنیم. این قانون ساده‌ترین قانون در میان بقیه‌ی قوانین محاسبه‌ی Big-O است. طبق این قانون باید به طور ساده از هر ثابت عددی غیر مرتبط با تعداد ورودی صرفنظر کنید. در واقع در Big-O در مقادیر بسیار بالای ورودی می‌توان از ضرایب چشم‌پوشی کرد. در نتیجه مهمترین قانون Big-O چنین است:

اگر f(n) برابر O(g(n)) باشد در این صورت kf(n) نیز به ازای k > 0 برابر O(g(n)) خواهد بود.

این یعنی پیچیدگی 5f(n) و f(n) هر دو برابر O(f(n)) است. در زیر مثالی از کُدی با پیچیدگی O(n) را نشان داده‌ایم.

```
function a(n) {
   var count = 0;

   for (var i = 0; i < n; i++) {
      count += 1;
   }

   return count;
}
```

در این مثال f(n) برابر n است. زیرا دارد کاری را n بار انجام می‌دهد. در نتیجه پیچیدگی زمانی تابع a() در این مثال برابر O(n) است. حال به کُد زیر توجه کنید:

```
function a(n) {
   var count = 0;

   for (var i = 0; i < 5 * n; i++) {
      count += 1;
   }

   return count;
}
```

در این مثال f(n) = 5n است. زیرا حلقه‌ی تابع a() از صفر تا 5n تکرار می‌شود. با این حال پیچیدگی زمانی این مثال نیز مانند مثال پیش برابر O(n) است. به بیان ساده علتش این است که اگر n به سمت بینهایت یا یک عدد بسیار بزرگ میل کند چهار کار اضافی‌ای که انجام می‌شود بی‌اهمیت خواهد بود. عملیاتی که n بار تکرار می‌شود به اندازه‌ی کافی بزرگ است. در نتیجه از هر ثابتی در محاسبه‌ی Big-O صرفنظر می‌شود.

کُد زیر تابع دیگری با پیچیدگی زمانی خطی را نشان می‌دهد که مانند مثال‌های پیشین است اما کاری را هم پس از پایان حلقه انجام می‌دهد (خط ۶).

```
function a(n) {
   var count = 0;

   for (var i = 0; i < n; i++) {
      count += 1;
   }

   count += 3;

   return count;
}
```

در این کُد f(n) = n + 1 است. این تابع نیز از نوع O(n) است. زیرا عملیات اضافی آن به تعداد ورودی n وابسته نیست. هنگامی که n به سمت بینهایت میل می‌کند، عملیات اضافی مزبور بی‌اهمیت و قابل چشم‌پوشی خواهد بود.

قانون جمع: O ها را با هم جمع کنید

قانون جمع بسیار ساده بوده و از نامش مشخص است چه معنی‌ای می‌دهد: پیچیدگی‌های زمانی را می‌شود با هم جمع کرد. تصور کنید الگوریتمی از دو الگوریتم دیگر تشکیل شده باشد. پیچیدگی زمانی این الگوریتم به طور ساده برابر جمع پیچیدگی زمانی دو الگوریتم تشکیل دهنده‌ی آن است.

اگر f(n) برابر O(h(n)) و g(n) برابر O(p(n)) باشد در این صورت f(n)+g(n) برابر O(h(n)+p(n)) خواهد بود.

تنها نکته‌ی مهمی که باید در نظر داشت این است که قانون ضریب باید پس از این قانون اعمال شود. کُد زیر تابعی با دو حلقه را نشان می‌دهد که پیچیدگی زمانی آنها مستقل از هم است. پیچیدگی زمانی این تابع برابر جمع پیچیدگی زمانی دو حلقه‌ی مزبور است:

```
function a(n) {
   var count = 0;

   for (var i = 0; i < n; i++) {
      count += 1;
   }

   for (var i = 0; i < 5 * n; i++) {
      count += 1;
   }

   return count;
}
```

در اینجا در خط چهارم f(n)=n و در خط هفتم f(n)=5n است. جمع این دو تابع برابر 6n خواهد بود. با اعمال قانون ضریب در نهایت پیچیدگی زمانی تابع a() برابر O(n) = n خواهد شد.

قانون ضرب: O ها را در هم ضرب کنید.

قانون ضرب به طور ساده بیان می‌کند O ها را می‌توان در هم ضرب کرد.

اگر f(n) برابر O(h(n)) و g(n) برابر O(p(n)) باشد در این صورت f(n).g(n) برابر O(h(n).p(n)) خواهد بود.

کُد زیر تابعی با دو حلقه‌ی تو در تو را نشان می‌دهد که قانون ضرب در آن قابل استفاده است:

```
function (n) {
   var count =0;

   for (var i = 0; i < n; i++) {
      count += 1;

      for (var i = 0; i < 5 * n; i++) {
         count += 1;
      }
   }

   return count;
}
```

در اینجا f(n) = 5n * n است زیرا خط هفتم 5n بار به ازای n عمل تکرار می‌شود. در نتیجه در کل $5n^2$ عملیات انجام خواهد شد. با اعمال قانون ضریب نتیجه‌ی نهایی برابر $O(n)=n^2$ خواهد بود.

قانون چند جمله‌ای: O توان k

قانون چند جمله‌ای بیان می‌کند پیچیدگی زمانی یک چند جمله‌ای برابر O ای از همان درجه‌ی چند جمله‌ای است. به صورت ریاضی این قانون چنین بیان می‌شود:

اگر f(n) یک چند جمله‌ای درجه k باشد در این صورت f(n) از نوع $O(n^k)$ است.

کُد زیر تنها از یک حلقه تشکیل می‌شود اما پیچیدگی زمانی درجه دو (مربعی) دارد.

```
function a(n) {
   var count = 0;

   for (var i = 0; i < n * n; i++) {
      count + =1;
   }

   return count;
}
```

در اینجا f(n) = n^2 است زیرا خط چهارم n * n بار تکرار می‌شود.

این مرور سریعی بر Big-O بود. در طول کتاب نیز مطالب بیشتری در این خصوص بیان خواهیم کرد.

خلاصه

نماد Big-O در تحلیل و مقایسه‌ی کارایی الگوریتم‌ها اهمیت دارد. تحلیل Big-O با نگاه به کُد و اعمال قوانینی برای ساده کردن Big-O شروع می‌شود. موارد زیر قوانین عمومی محاسبه‌ی O است:

- ضرایب و ثابت‌ها را حذف کنید (قانون ضریب)
- O ها را جمع کنید (قانون جمع)
- O ها را ضرب کنید (قانون ضرب)
- O یک چند جمله‌ای درجه n، به توان درجه‌ی چند جمله‌ای است (قانون چند جمله‌ای)

تمرین

پیچیدگی زمانی کُدهای زیر را به دست بیاورید.

تمرین ۱:

```
function someFunction(n) {
   for (var i = 0; i < n * 1000; i++) {
      for (var j = 0; j < n * 20; j++) {
         console.log(i + j);
      }
   }
}
```

تمرین ۲:

```
function someFunction(n) {
   for (var i = 0; i < n; i++) {
      for (var j = 0; j < n; j++) {
         for (var k = 0; k < n; k++) {
            for (var l = 0; l < 10; l++) {
               console.log(i + j + k + l);
            }
         }
      }
   }
}
```

تمرین ۳:

```
function someFunction(n) {
   for (var i = 0; i < 1000; i++) {
      console.log("hi");
   }
}
```

تمرین ۴:

```
function someFunction(n) {
   for (var i = 0; i < n * 10; i++) {
      console.log(n);
   }
}
```

تمرین ۵:

```
function someFunction(n) {
   for (var i = 0; i < n; i * 2) {
      console.log(n);
   }
}
```

تمرین ۶:

```
function someFunction(n) {
   while (true){
      console.log(n);
   }
}
```

پاسخ:

۱. $O(n^2)$
دو حلقه به صورت تو در تو داریم. از ضریب جلوی n صرفنظر می‌شود.

۲. $O(n^3)$
چهار حلقه‌ی تو در تو داریم، اما آخرین حلقه ۱۰ بار بیشتر اجرا نمی‌شود.

۳. $O(1)$
این کُد پیچیدگی زمانی ثابتی دارد. زیرا تابع از صفر تا ۱۰۰۰ اجرا شده و به n وابستگی ندارد.

۴. $O(n)$
این کُد پیچیدگی زمانی خطی دارد. زیرا تابع از 0 تا n * 10 اجرا می‌شود. از ثوابت هم صرفنظر می‌شود.

۵. $O(\log_2 n)$
این کُد پیچیدگی زمانی لگاریتمی دارد. زیرا به ازای ورود n این کُد $\log_2 n$ بار اجرا می‌شود علتش این است که i بجای این که مانند دیگر مثال‌ها یک بار زیاد شود هر بار دو بار افزایش داده می‌شود.

۶. $O(\infty)$
پیچیدگی زمانی این کُد بینهایت است، زیرا تابع هیچگاه پایان نمی‌پذیرد.

فصل ۲. نکته‌های مهم زبان جاوااسکریپت

در این فصل به اختصار برخی از قواعد نحوی و رفتار ویژه‌ی زبان جاوااسکریپت را بیان می‌کنیم. به عنوان یک زبان تفسیری داینامیک قوانین نحوی جاوااسکریپت با زبان‌های شی‌گرای سنتی فرق دارد. درک مفاهیم این فصل برای استفاده‌ی درست از جاوااسکریپت بسیار مهم است و به شما کمک می‌کند درک بهتری نسبت به طراحی و پیاده‌سازی الگوریتم‌های مختلف به زبان جاوااسکریپت پیدا کنید.

حوزه‌ی دید

حوزه‌ی دید چیزی است که دسترسی جاوااسکریپت به متغیرها را تعریف می‌کند. در جاوااسکریپت متغیرها می‌توانند به حوزه‌ی دید سراسری (global) یا محلی (local) تعلق داشته باشند. متغیرهای سراسری متغیرهایی هستند که در حوزه‌ی دید سراسری قرار داشته و از هر جایی در برنامه قابل دسترس هستند.

تعریف متغیر به صورت سراسری: حوزه‌ی دید سراسری

در جاوااسکریپت می‌توانید متغیرها را بدون استفاده از هیچ عملگری تعریف کرده و استفاده کنید. به مثال زیر توجه کنید:

```
test = "sss";
console.log(test); // prints "sss"
```

در اینجا بدون آن که متغیر test را به صورت صریح از پیش تعریف کرده باشیم آن را مقداردهی کرده و چاپ می‌کنیم. این کار باعث می‌شود متغیر test به صورت سراسری تعریف شود. این یکی از بدترین کارهایی است که در جاوااسکریپت می‌شود کرد و باید همیشه به هر بهایی که شده از آن اجتناب کنید. همیشه به صورت صریح متغیرها را با استفاده از var یا let تعریف کنید. اگر هم متغیری دارید که قرار نیست مقدارش بعدا تغییر کند آن را با const تعریف کنید.

متغیرهای var: حوزه‌ی دید تابعی

در جاوااسکریپت از کلمه‌ی کلیدی var برای تعریف متغیر استفاده می‌شود. تعریف چنین متغیرهایی به صورت شناور انجام می‌شود. یعنی جاوااسکریپت هنگام برخورد با خط تعریف چنین متغیری تعریف آن را آنقدر رو به بالا جابجا می‌کند تا در نهایت به ابتدای تابعی که متغیر داخل آن تعریف شده برسد. به چنین عملی variable hoisting یا برافراشتن متغیر گفته می‌شود. به مثال زیر توجه کنید:

```
function scope1() {
   var top = "top";
   bottom = "bottom";
   console.log(bottom);

   var bottom;
}

scope1(); // prints "bottom" - no error
```

در اینجا متغیر bottom با این که در انتهای تابع scope1() تعریف شده بدون هیچ خطایی در خط ۳ مقداردهی و در خط ۴ نیز مقدارش چاپ می‌شود. زیرا جاوااسکریپت پیش از اجرای تابع scope1() خط تعریف متغیر bottom (خط ۶) را بالا آورده و به ابتدای تابع منتقل می‌کند. کُد بالا درست مانند کُد زیر است:

```
function scope1() {
   var top = "top";
   var bottom;

   bottom = "bottom";
   console.log(bottom);
}

scope1(); // prints "bottom" - no error
```

نکته‌ی کلیدی var که باید در نظر داشته باشید این است که حوزه‌ی دید متغیرهایی که با var تعریف می‌شود برابر نزدیک‌ترین یا درونی‌ترین تابعی است که متغیر داخل آن قرار گرفته. یعنی چه؟ بگذارید مثال دیگری بزنیم.

در کُد زیر تابع scope2() نزدیک‌ترین تابع در رابطه با متغیر print است.

```
function scope2(print) {
   if (print) {
      var insideIf = '12';
   }

   console.log(insideIf);
}

scope2(true); // prints '12' - no error
```

در اینجا با وجودی که متغیر insideIf داخل if تعریف شده اما دستور console.log() بیرون if بدون هیچ خطایی قادر است آن را چاپ کند. زیرا جاوااسکریپت پیش از اجرای تابع scope2() هنگامی که آن را پردازش کرده و با خط تعریف متغیر insideIf (خط ۳) مواجه می‌شود حوزه‌ی دید آن را برابر تابع scope2() قرار می‌دهد. به همین دلیل است که متغیر insideIf بیرون if هم قابل مشاهده است. در واقع کُد بالا مانند کُد زیر است:

```
function scope2(print) {
   var insideIf;

   if (print) {
      insideIf = '12';
   }

   console.log(insideIf);
}

scope2(true); // prints '12' - no error
```

اگر بخواهید کُد قبلی را در زبانی مانند جاوا اجرا کنید اصلا کامپایل نمی‌شود. زیرا متغیر insideIf تنها داخل همان دستور if که در آن تعریف شده قابل مشاهده خواهد بود نه بیرون آن.

مثالی دیگر:

```
var a = 1;

function foo() {
   if (true) {
      var a = 4;
   }

   console.log(a); // prints '4'
}

foo();
```

در اینجا در خروجی برای a عدد 4 چاپ می‌شود، زیرا متغیر a که در console.log() چاپ شده متغیری است که داخل دستور if تعریف شده. بار دیگر با وجودی که متغیر a داخل دستور if تعریف شده اما جاوااسکریپت آن را در حوزه‌ی دید تابع foo قرار می‌دهد و هنگامی که console.log(a) قرار است اجرا شود از آنجایی که در حوزه‌ی دید تابع foo متغیری به نام a وجود دارد مقدار همین متغیر چاپ می‌شود نه متغیر سراسری a که بیرون تابع foo() تعریف شده.

توجه کنید جاوااسکریپت تنها تعریف متغیر را به ابتدای تابع منتقل می‌کند. این مساله به این معنی نیست که خط تعریف متغیر را در سورس تابع بالا آورده و آن را در ابتدا اجرا می‌کند. به مثال زیر توجه کنید:

```
function foo() {
   console.log('foo');
   return 10;
}
function bar() {
   console.log('bar');
   return 20;
}
function baz() {
   console.log('baz');
   return 30;
}
function scopes(print) {
   var a = foo();
   if (print) {
      var b = bar();
   }
   console.log(a);
   console.log(b);
   console.log(c);
   console.log(d);

   var c = baz(), d = 40;
   return '';
}

scopes(true);
```

خروجی این کُد به صورت زیر است:

```
foo
bar
10
20
undefined
undefined
baz
```

همان گونه که می‌بینید با وجودی که جاوااسکریپت متغیرهای b، c و d را hoist کرده و تعریف آنها را بالا می‌آورد اما خط آنها واقعا همان جایی که هست به همان ترتیبی که در سورس تابع scopes() ذکر شده اجرا می‌شود.

در واقع جاوااسکریپت با وجودی که تعریف متغیر را بالا می‌آورد اما حواسش هست که سورس را دست نزده و خراب نکند. به همین دلیل وقتی دستور console.log(c) و console.log(d) اجرا می‌شود مقدار undefined برای متغیرهای c و d چاپ می‌شود، زیرا خط مقداردهی این متغیرها هنوز اجرا نشده و این متغیرها هنوز مقداری ندارند. خط 23 نیز واقعا در انتهای اجرای تابع scopes() چاپ می‌شود.

متغیرهای let: حوزه‌ی دید بلاکی

کلمه‌ی کلیدی دیگری که با آن می‌توانید متغیر تعریف کنید let است. متغیرهایی که با let تعریف می‌شوند حوزه‌ی دید بلاکی دارند و تنها در نزدیک‌ترین بلاک کُدی که داخل آن قرار گرفته‌اند (داخل نزدیک‌ترین { }) قابل مشاهده هستند.

```
function scope3(print) {
   if (print) {
      let insideIf = '12';
   }
   console.log(insideIf);
}

scope3(true); // prints ''
```

در اینجا چیزی در خروجی چاپ نمی‌شود، زیرا متغیر insideIf تنها داخل دستور if قابل مشاهده است.

مقایسه‌ی برابری

نوع داده‌های زبان جاوااسکریپت با دیگر زبان‌های برنامه‌نویسی فرق دارد. این مساله بر انجام کارهای مختلف با متغیرها مانند مقایسه‌ی برابری تاثیرگذار است. در این قسمت این مساله را بررسی می‌کنیم. پیش از آن بگذارید ابتدا مروری بر انواع متغیر در زبان جاوااسکریپت داشته باشیم.

انواع متغیر

در جاوااسکریپت هفت نوع داده‌ی پایه وجود دارد: boolean، number، string، object، undefined، function و symbol (در اینجا کاری با نوع symbol نداریم). از بین این نوع داده‌ها undefined نوع ویژه‌ای است که به

متغیرهای بدون مقدار نسبت داده می‌شود. به عبارت دیگر نوع متغیری که هنوز چیزی به آن نسبت داده نشده undefined یا «تعریف نشده» است.

برای به دست آوردن نوع متغیرها از عملگر typeof استفاده می‌شود.

```
var is20 = false;
var age = 19;
var lastName = "Bae";
var fruits = ["Apple", "Banana", "Kiwi"];
var me = { firstName:"Sammie", lastName:"Bae" };
var nullVar = null;
var function1 = function(){ }
var blank;

typeof is20;        // boolean
typeof age;         // number
typeof lastName;    // string
typeof fruits;      // object
typeof me;          // object
typeof nullVar;     // object
typeof function1;   // function
typeof blank;       // undefined
```

بررسی درستی (truthy) و نادرستی (falsey)

کاربرد ارزیابی یا بررسی درست و نادرست بودن عبارت‌ها در دستورهای شرطی مانند if، while یا for است. در بسیاری از زبان‌ها عبارتی که به این دستورها داده می‌شود باید یک عبارت منطقی یا مقداری از نوع boolean باشد. اما جاوااسکریپت (و همچنین سایر زبان‌های داینامیک) در این خصوص انعطاف‌پذیرتر هستند. به مثال زیر توجه کنید:

```
if (node) {
   ...
}
```

در اینجا node یک متغیر است. اگر مقدار این متغیر برابر رشته‌ی تهی، null یا undefined باشد به عنوان false تعبیر می‌شود. مقادیری که به عنوان false ارزیابی می‌شوند چنین هستند:

- false
- 0
- رشته‌ی تهی (’’ و "")
- NaN
- undefined
- null

از آن سو مقادیری که به عنوان true ارزیابی می‌شوند چنین هستند:

- true
- هر عددی به جز 0
- رشته‌های غیر تهی

- توابع
- اشیا

مثال:

```
var printIfTrue = '';

if (printIfTrue) {
   console.log('truthy');
} else {
   console.log('falsey'); // prints 'falsey'
}
```

عملگر === و ==

جاوااسکریپت یک زبان اسکریپتی است و نوع متغیرها در طول اجرای برنامه می‌تواند تغییر کند. از این رو هنگام تعریف متغیرها تا زمانی که مقداری برایشان مشخص نشود اصلا نوعی به آنها نسبت داده نشده و نوع متغیرها در حین اجرای کُد بررسی می‌شود.

برای بررسی برابری دو مقدار از عملگر == استفاده می‌شود. اما این عملگر نه به معنی برابر بودن بلکه به معنی *قابل برابر بودن* است. به همین دلیل عبارت true == 1 معادل true ارزیابی می‌شود، اما به این معنی نیست که 1 و true یک چیز هستند. برای بررسی این که برابری دو عبارت، دو متغیر یا دو مقدار به معنی واقعی انجام شود، یعنی نوع آنها نیز با هم برابر باشد، باید از عملگر === استفاده کنیم. در این حالت دیگر true === 1 معادل true نخواهد بود.

```
'5' == 5 // returns true
'5' === 5 // returns false
```

در اینجا مقدار عبارت اول true است زیرا پیش از انجام عمل مقایسه، '5' به عدد 5 تبدیل می‌شود. اما مقدار عبارت دوم false است زیرا '5' یک رشته و 5 یک عدد است.

مقایسه‌ی اشیا

در جاوااسکریپت اشیا با استفاده از آکلاد تعریف می‌شوند.

```
var me = { firstName:"Sammie", lastName:"Bae" };
```

در جاوااسکریپت هنگام مقایسه‌ی اشیا با استفاده از عملگر == یا =! ارجاع آنها با هم مقایسه می‌شود. چنانچه ارجاع دو متغیر یکسان باشد (هر دو به یک شی در حافظه ارجاع کنند) با هم برابر تلقی شده و در غیر این صورت نابرابر خواهند بود.

مثال:

```
var obj1 = { name: "John Doe" }
var obj2 = { name: "John Doe" }
var obj3 = obj1;

console.log(obj1 == obj2) // false
```

```
console.log(obj1 == obj3) // true
```

در اینجا با وجودی که دو شی obj1 و obj2 همانند هم هستند اما با هم برابر نیستند. زیرا آدرس‌های متفاوتی در حافظه دارند (دو شی متفاوت در حافظه هستند). اما obj3 با obj1 برابر است. زیرا به همان شی ارجاع می‌کند.

چنانچه مقصودمان از برابر بودن دو متغیر که حاوی شی هستند برابر بودن خصوصیت‌هایشان باشد باید خودمان این بررسی را انجام داده و مثلا تابعی برای آن بنویسیم. این کار در کتابخانه‌هایی مانند loadash [۱] یا underscore [۲] انجام شده است. این دو کتابخانه تابعی به نام isEqual(obj1, obj2) دارند که برای مقایسه‌ی برابری دو شی یا دو مقدار به کار می‌رود. الگوریتم مقایسه‌ی برابری تابع isEqual() بدین صورت است که در یک حلقه خصوصیت‌های دو شی را با هم مقایسه می‌کند که آیا برابر هستند یا خیر.

برای نمونه در زیر یک تابع مقایسه کننده‌ی برابری ساده نشان داده شده است:

```
function isEquivalent(a, b) {
   // arrays of property names
   var aProps = Object.getOwnPropertyNames(a);
   var bProps = Object.getOwnPropertyNames(b);

   // If their property lengths are different, they're different objects
   if (aProps.length != bProps.length) {
      return false;
   }

   for (var i = 0; i < aProps.length; i++) {
      var propName = aProps[i];

      // If the values of the property are different, not equal
      if (a[propName] !== b[propName]) {
         return false;
      }
   }

   // If everything matched, correct
   return true;
}

isEquivalent({'hi':12},{'hi':12}); // returns true
```

همان گونه که گفتیم این تابع پیاده‌سازی ساده‌ای از الگوریتم مقایسه‌ی برابری است و تنها برای اشیائی جواب می‌دهد که خصوصیت‌هایشان از نوع عدد یا رشته باشد. چنانچه حداقل یکی از خصوصیت‌های اشیاء مورد مقایسه از نوع شی، آرایه یا تابع باشد این تابع دوباره مقدار false برمی‌گرداند.

```
var obj1 = { prop1: 'test', prop2: function() { } };
var obj2 = { prop1: 'test', prop2: function() { } };
```

[1] https://lodash.com/
[2] http://underscorejs.org/

```
isEquivalent(obj1,obj2); // returns false
```

با وجودی که تابع‌های مشخص شده برای خصوصیت‌های calc شبیه هم هستند اما با هم برابر نیستند. به مثال زیر توجه کنید:

```
var function1 = function() { console.log(2) };
var function2 = function() { console.log(2) };

console.log(function1 == function2); // prints 'false'
```

این دو تابع یک کار انجام می‌دهند، اما در عمل دو تابع متفاوت بوده و آدرس‌شان در حافظه با هم متفاوت است. از این رو عملگر == و === برای آنها false برمی‌گرداند. به طور کلی آنچه باید به خاطر بسپارید این است که عملگرهای == و === (یا =! و ==!) را تنها برای رشته، عدد و مقادیر boolean باید به کار ببرید. این عملگرها در مقایسه‌ی اشیاء و توابع، ارجاع آنها را چک کرده و بررسی می‌کنند هر دو به یک آدرس در حافظه ارجاع می‌کنند یا خیر. اگر مقصودتان از برابر بودن دو شی برابر بودن خصوصیت‌هایشان باشد برای مقایسه بجای == یا === از یک تابع شخصی مانند ()isEquivalent استفاده کنید.

خلاصه

جاوااسکریپت شیوه‌ی متفاوتی برای تعریف متغیرها نسبت به دیگر زبان‌های برنامه‌نویسی دارد. کلمه‌ی کلیدی var متغیر را در حوزه‌ی دید تابعی و کلمه‌ی کلیدی let متغیر را در حوزه‌ی دید بلاکی تعریف می‌کند. بدون استفاده از این دو کلمه‌ی کلیدی نیز می‌توانید هر متغیری را در حوزه‌ی دید سراسری تعریف کنید. اما از این کار همیشه اجتناب کنید.

برای به دست آوردن نوع متغیر از عملگر typeof استفاده می‌شود. در نهایت برای انجام مقایسه‌ی برابری اگر هدف‌تان بررسی برابری مقادیر است از == و اگر بررسی برابری مقادیر و همچنین نوع آنها است از === استفاده کنید. با این وجود دو عملگر یاد شده را تنها برای رشته، عدد و boolean به کار ببرید. برای بررسی برابری اشیا باید از یک الگوریتم شخصی استفاده کنید.

فصل ۳. اعداد در جاوااسکریپت

در این فصل به اعداد، عملیات روی اعداد، نحوه‌ی نمایش اعداد، تابع Number، الگوریتم‌های رایج عددی و تولید اعداد تصادفی در جاوااسکریپت می‌پردازیم. در انتهای این فصل نحوه‌ی کار با اعداد در جاوااسکریپت و همچنین نحوه‌ی به دست آوردن اعداد اول تشکیل دهنده‌ی یک عدد را یاد می‌گیرید که مساله‌ی بسیار مهمی در الگوریتم‌های رمزنگاری است.

عملگرهای عددی زبان‌های برنامه‌نویسی امکان انجام عملیات ریاضی و محاسبه‌ای را فراهم می‌کنند. عملگرهای ریاضی زبان جاوااسکریپت چنین هستند:

+: جمع

-: تفریق

/: تقسیم

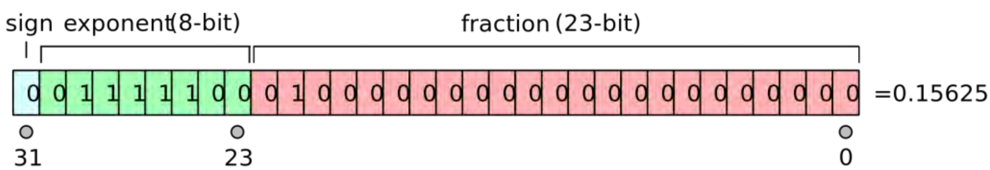
*: ضرب

%: باقیمانده

این عملگرها در دیگر زبان‌های برنامه‌نویسی هم وجود دارند و مخصوص جاوااسکریپت نیستند.

سیستم عددی

همان گونه که در شکل ۱-۳ نشان داده شده جاوااسکریپت برای نمایش اعداد از نمایش ۳۲ بیتی نقطه شناور استفاده می‌کند. مقدار نمایش داده شده در این مثال برابر 0.15625 است. اگر بیت علامت (۳۱ اُمین بیت) برابر 1 باشد نشان می‌دهد عدد منفی است. ۸ بیت بعدی (بیت ۳۰ اُم تا ۲۳ اُم) قسمت صحیح و ۲۳ بیت بعدی برای نگهداری قسمت اعشار یا کسری عدد به کار می‌رود.



شکل ۱-۳. سیستم عددی نقطه شناور ۳۲ بیتی

عددی که با این ۳۲ بیت ذخیره می‌شود بر اساس فرمول عجیب و غریب زیر محاسبه می‌شود:

$$\text{value} = (-1)^{\text{sign}} \times 2^{\text{e}-127} \times \left(1 + \sum_{t=1}^{23} b_{23-t}\, 2^{-t}\right)$$

جزییات محاسبه‌ی عدد 0.15625 شکل ۱–۳ بدین صورت است:

$$\text{sign} = 0$$

$$e = (0111100)_2 = 124 \text{ (in base 10)}$$

$$1+\sum_{i=1}^{23} b_{23-i}\, 2^{-i} = 1+0+0.25+0$$

نتیجه‌ی این عبارت چنین خواهد بود:

$$\text{value} = 1 \times 2^{124-127} \times 1.25 = 1 \times 2^{-3} \times 1.25 = 0.15625$$

البته این سیستم نقطه شناور برای اعداد کسری کمی خطای محاسباتی دارد. برای نمونه اعداد 0.1 و 0.2 را نمی‌توان به صورت دقیق ذخیره کرد. در نتیجه 0.3 === 0.2 + 0.1 برابر true نیست.

```
0.1 + 0.2 === 0.3; // prints 'false'
```

برای درک این که چرا واقعا نمی‌توان 0.1 را به طور دقیق به صورت یک عدد ۳۲ بیتی نقطه شناور ذخیره کرد باید سیستم دودویی را بفهمید. نمایش بخش اعشاری به صورت دودویی به تعداد بی‌انتهایی رقم یا بیت نیاز دارد. زیرا اعداد دودویی به صورت 2^n نمایش داده می‌شوند که n در آن یک عدد صحیح است.

```
          0.0011...
        -----------
1010 |    1.00000
          0
          __
          10
           0
          ___
          100           a)
            0
          ___
          1000
             0
          _____
          10000
           1010
          _____
            1100
            1010
            ____
              100       repeated in a)
```

شکل ۲–۳. تقسیم بی‌انتها برای محاسبه‌ی 0.1 در مبنای دو

برای تبدیل 0.1 به مبنای دو عمل تقسیم هیچگاه خاتمه پیدا نمی‌کند. همان گونه که در شکل ۲–۳ نشان داده شده معادل دودویی عدد 10 برابر 1010 است. محاسبه‌ی 0.1 یا 1/10 به دنباله‌ی بی‌انتهایی از عمل تقسیم و تولید بی‌انتهای بخش اعشاری منجر می‌شود.

شی Number

در جاوااسکریپت شی‌ای به نام Number وجود داشته و خصوصیت‌هایی دارد که به رفع مشکل بالا کمک می‌کند.

تقسیم صحیح و رُند کردن

از آنجایی که جاوااسکریپت اعداد را به صورت اعشاری نقطه شناور ذخیره می‌کند در جاوااسکریپت تقسیم صحیح وجود ندارد. در دیگر زبان‌های برنامه‌نویسی مانند جاوا هنگام تقسیم صحیح تنها از پیمانه استفاده می‌شود. برای نمونه حاصل 5/4 در جاوا برابر 1 است زیرا پیمانه این تقسیم برابر 1 است (از باقیمانده‌ی 1 هم صرفنظر می‌شود). با این وجود حاصل 5/4 در جاوااسکریپت برابر عدد صحیح 1 نبوده و یک عدد اعشاری است.

```
5 / 4    // 1.25
```

از آنجایی که در زبان‌های دیگر مانند جاوا اعداد صحیح هم وجود دارد هنگام تعریف متغیر یا انجام عمل تقسیم از شما خواسته می‌شود به طور صریح مشخص کنید عدد یا عملیات تقسیمی که انجام می‌دهید چه نوعی است. برای این که در جاوااسکریپت بتوانید عمل تقسیم صحیح انجام بدهید باید پس از تقسیم، عدد به دست آمده را با استفاده از یکی از توابع زیر رُند کنید:

- Math.floor: رُند رو به پایین
- Math.round: رُند به نزدیک‌ترین عدد
- Math.ceil: رُند رو به بالا

مثال:

```
Math.floor(0.9);   // 0
Math.floor(1.1);   // 1
Math.round(0.49);  // 0
Math.round(0.5);   // 1
Math.round(2.9);   // 3
Math.ceil(0.1);    // 1
Math.ceil(0.9);    // 1
Math.ceil(21);     // 21
Math.ceil(21.01);  // 22
```

Number.EPSILON

شی Number خصوصیتی به نام EPSILON دارد که برابر کوچکترین عدد قابل ثبت بین دو عدد در زبان جاوااسکریپت است. این مقدار برای رفع مشکل تقریب اعداد اعشاری مفید است. برای نمونه برای این که ببینیم دو عدد یا دو عبارت عددی با هم برابر هستند می‌توانیم تابعی به صورت زیر بنویسیم:

```
function numberEquals(x, y) {
   return Math.abs(x - y) < Number.EPSILON;
}
```

این تابع برای مقایسه‌ی عبارت 0.2 + 0.1 و عدد 0.3 مقدار true بر می‌گرداند.

```
numberEquals(0.1 + 0.2, 0.3); // true
```

کاری که در تابع ()numberEquals انجام می‌دهیم این است که حاصل تفریق عبارت‌های داده شده را حساب می‌کنیم. اگر قدر مطلق نتیجه‌ی به دست آمده از Number.EPSILON کوچکتر باشد می‌توانیم ادعا کنیم دو عبارت با هم برابر هستند. فراموش نکنید Number.EPSILON کوچکترین اختلاف عددی قابل ثبت بین دو عدد است.

ماگزیمم

خصوصیت Number.MAX_SAFE_INTEGER بزرگ‌ترین عدد صحیح موجود در جاوااسکریپت را برمی‌گرداند. از آنجایی که بزرگ‌تر از این مقدار، عددی در جاوااسکریپت وجود ندارد نتیجه‌ی مقایسه‌ی زیر true است:

```
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2; // true
```

با این حال مراقب باشید این مقایسه برای اعداد اعشاری false می‌شود.

```
Number.MAX_SAFE_INTEGER + 1.111 === Number.MAX_SAFE_INTEGER + 2.022; // false
```

خصوصیت دیگری به نام Number.MAX_VALUE هم وجود دارد که بزرگ‌ترین عدد اعشاری موجود در جاوااسکریپت را برمی‌گرداند که برابر 1.7976931348623157e+308 است. اگر عمل مقایسه‌ی عبارت‌های قبلی را با Number.MAX_VALUE انجام بدهیم این بار برای اعداد اعشاری نتیجه‌ی true خواهیم گرفت.

```
Number.MAX_VALUE + 1 === Number.MAX_VALUE + 2; // true
Number.MAX_VALUE + 1.111 === Number.MAX_VALUE + 2.022; // true
```

می‌نیمم

مشابه Number.MAX_VALUE و Number.MAX_SAFE_INTEGER دو خصوصیت Number.MIN_VALUE و Number.MIN_SAFE_INTEGER هم وجود دارد که به ترتیب برابر کوچکترین عدد اعشاری و کوچکترین عدد صحیح موجود در زبان جاوااسکریپت است. Number.MIN_SAFE_INTEGER برابر 9007199254740991- است. از آنجایی که کوچکتر از این مقدار عدد صحیحی در جاوااسکریپت وجود ندارد نتیجه‌ی مقایسه‌ی زیر true است:

```
Number.MIN_SAFE_INTEGER - 1 === Number.MIN_SAFE_INTEGER - 2; // true
```

اما به طور مشابه این مقایسه برای اعداد اعشاری کار نمی‌کند.

```
Number.MIN_SAFE_INTEGER - 1.111 === Number.MIN_SAFE_INTEGER - 2.022; // false
```

مقدار Number.MIN_VALUE نیز برابر 5e-24 و کوچکترین عدد اعشاری نزدیک به صفر است (توجه کنید این عدد منفی نیست).

```
Number.MIN_VALUE - 1 == -1; // true
```

این مقایسه مشابه مقایسه‌ی 1- == 1 - 0 است.

از نظر مقایسه‌ای، Number.MIN_VALUE بزرگ‌تر از Number.MIN_SAFE_INTEGER است.

بینهایت یا infinity

تنها مقدار بزرگتر از Number.MAX_VALUE بینهایت یا Infinity است. تنها مقدار کوچکتر از Number.MAX_SAFE_INTEGER نیز -Infinity است.

```
Infinity > Number.MAX_SAFE_INTEGER; // true
-Infinity < Number.MAX_SAFE_INTEGER // true
-Infinity -32323323 == -Infinity -1; // true
```

علت true بودن این مقایسه‌ها این است که چیزی بزرگتر از Infinity یا کوچکتر از -Infinity وجود ندارد.

خلاصه‌ی بزرگی و کوچکی مقادیر ثابت جاوااسکریپت

در زیر لیست ثوابت عددی جاوااسکریپت را به ترتیب از کوچک (چپ) به بزرگ (راست) ذکر کرده‌ایم:

```
-Infinity    < Number.MIN_SAFE_INTEGER
             < Number.MIN_VALUE
             < 0
             < Number.MAX_SAFE_INTEGER
             < Number.MAX_VALUE
             < Infinity
```

الگوریتم‌های عددی

یکی از الگوریتم‌های مشهور کار با اعداد، بررسی اول بودن یک عدد است. اعداد اول پایه و اساس الگوریتم‌های رمزنگاری (فصل ۴) و هَش کردن (فصل ۱۱) را تشکیل می‌دهد. از این رو بد نیست چند الگوریتم مربوط به کار با اعداد اول را با هم ببینیم.

در ریاضیات به عددی اول گفته می‌شود که به جز خودش و عدد ۱ به هیچ عددی بخش‌پذیر نباشد. بگذارید الگوریتم بررسی اول بودن یک عدد را مرور کنیم.

آزمایش اول بودن

یک راه برای بررسی اول بودن عددی مانند n این است که اعداد 2 تا n را بر آن تقسیم کنیم. باقیمانده‌ی هیچ یک از این تقسیم‌ها نباید صفر باشد (n نباید به هیچ عددی به جز ۱ بخش‌پذیر باشد).

```
function isPrime(n){
   if (n <= 1) {
      return false;
   }

   // check from 2 to n-1
   for (var i = 2; i < n; i++) {
      if (n % i == 0) {
         return false;
      }
   }

   return true;
}
```