



<https://algs4.cs.princeton.edu>

## 1.4 ANALYSIS OF ALGORITHMS

---

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*



<https://algs4.cs.princeton.edu>

## 1.4 ANALYSIS OF ALGORITHMS

---

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

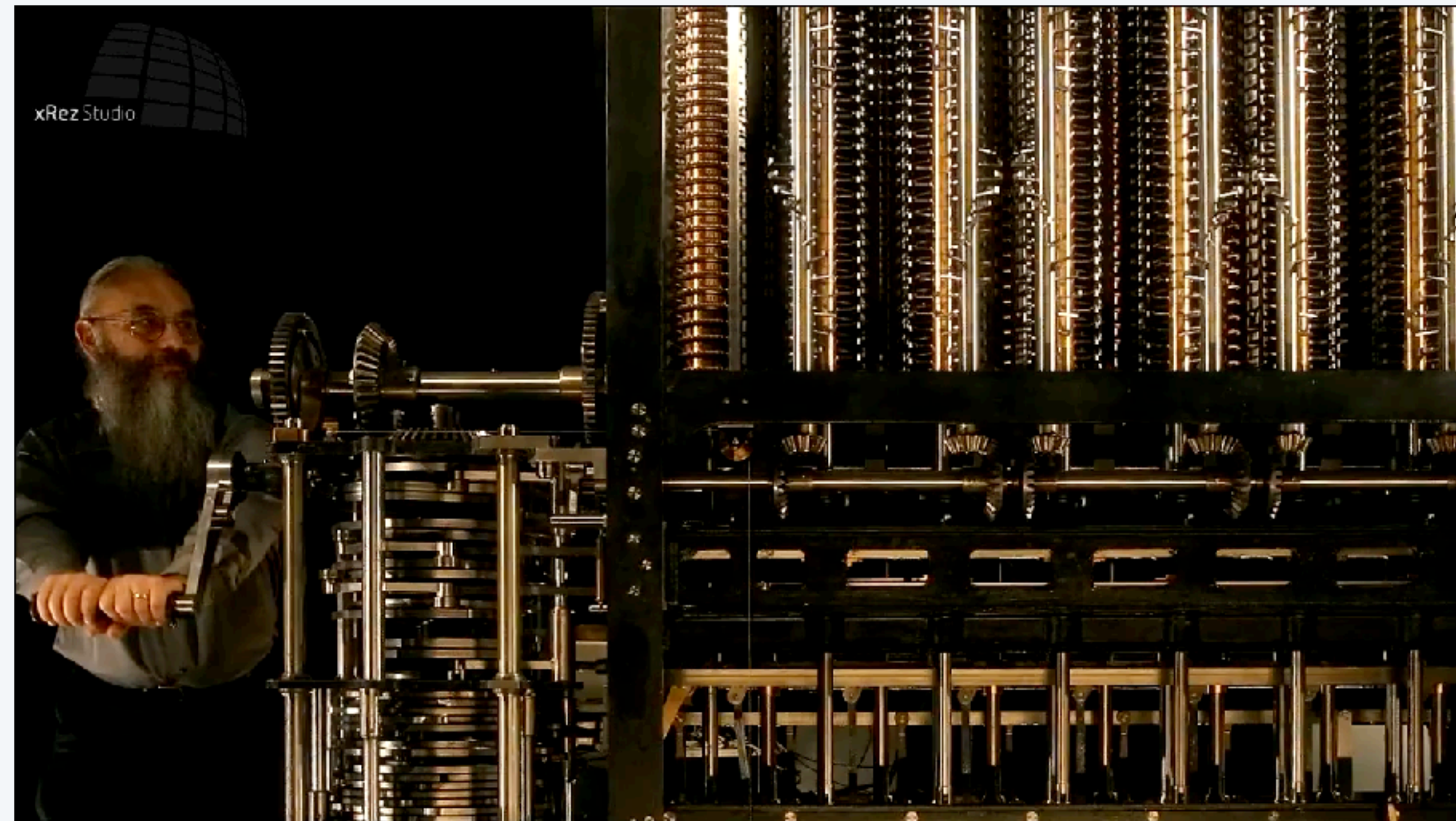
# Running time

---

*“ As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the **shortest time** ? ”* — Charles Babbage (1864)



*how many times  
do you have to turn  
the crank?*



<https://vimeo.com/49080293>

## Running time

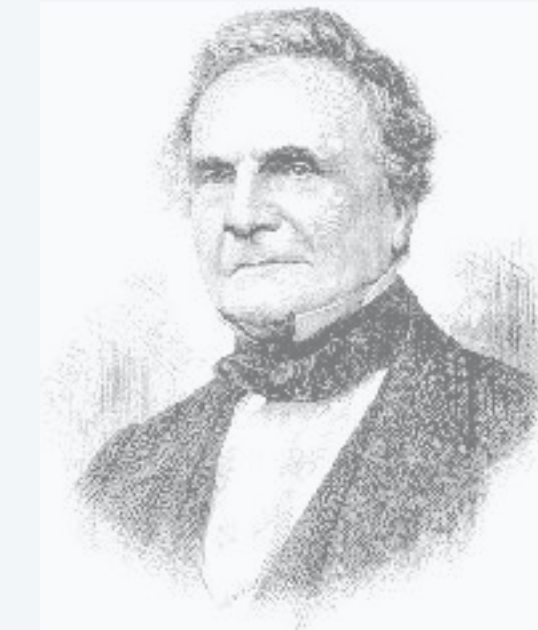


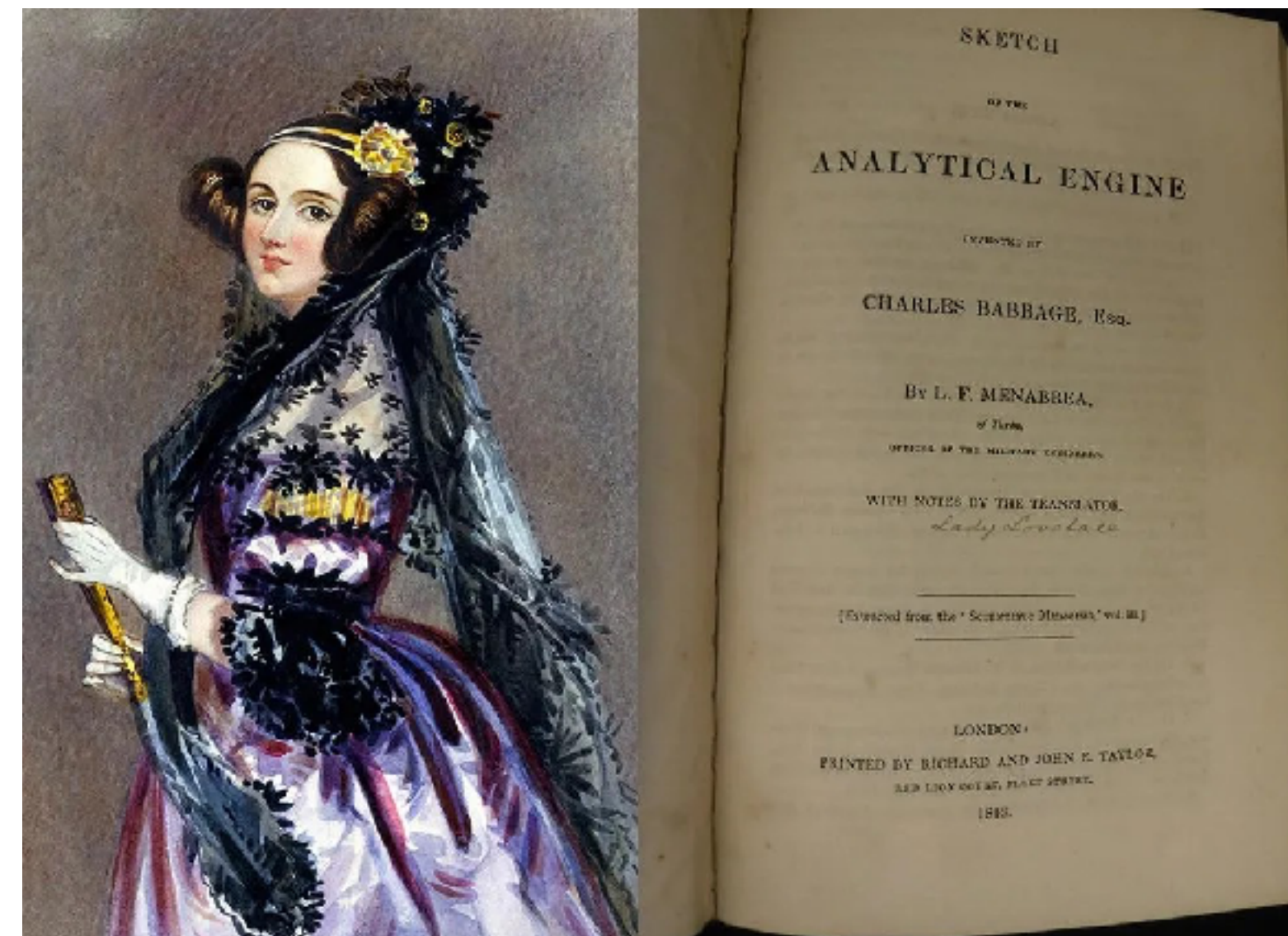
Diagram for the comparison of the Regions of the Numbers of Bernoulli. See Note G, page 729 of 1932.

Number of Operands Number of Operations	Variables and signs	Variables and signs	Relation of the change of the value of the Variable	Structure of the Result	Data					Working Variables					Result Variables				
					$U_1$	$U_2$	$U_3$	$U_4$	$U_5$	$W_1$	$W_2$	$W_3$	$W_4$	$W_5$	$W_6$	$W_7$	$W_8$	$W_9$	$W_{10}$
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
4	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
5	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
6	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
8	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
9	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
10	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
11	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
12	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
13	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
14	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
15	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
16	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
17	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
18	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
19	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
20	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
21	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
22	$\times$	$U_1 \times U_2$	$U_1 U_2$	$U_1 U_2$	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
23	$-$	$U_1 - U_2$	$U_1 - U_2$	$U_1 - U_2$	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
24	$+$	$U_1 + U_2$	$U_1 + U_2$	$U_1 + U_2$	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

How follows a symbol of Operation (1) or (2) or (3) or (4) or (5) or (6) or (7) or (8) or (9) or (10) or (11) or (12) or (13) or (14) or (15) or (16) or (17) or (18) or (19) or (20) or (21) or (22) or (23) or (24) or (25) or (26) or (27) or (28) or (29) or (30) or (31) or (32) or (33) or (34) or (35) or (36) or (37) or (38) or (39) or (40) or (41) or (42) or (43) or (44) or (45) or (46) or (47) or (48) or (49) or (50) or (51) or (52) or (53) or (54) or (55) or (56) or (57) or (58) or (59) or (60) or (61) or (62) or (63) or (64) or (65) or (66) or (67) or (68) or (69) or (70) or (71) or (72) or (73) or (74) or (75) or (76) or (77) or (78) or (79) or (80) or (81) or (82) or (83) or (84) or (85) or (86) or (87) or (88) or (89) or (90) or (91) or (92) or (93) or (94) or (95) or (96) or (97) or (98) or (99) or (100) or (101) or (102) or (103) or (104) or (105) or (106) or (107) or (108) or (109) or (110) or (111) or (112) or (113) or (114) or (115) or (116) or (117) or (118) or (119) or (120) or (121) or (122) or (123) or (124) or (125) or (126) or (127) or (128) or (129) or (130) or (131) or (132) or (133) or (134) or (135) or (136) or (137) or (138) or (139) or (140) or (141) or (142) or (143) or (144) or (145) or (146) or (147) or (148) or (149) or (150) or (151) or (152) or (153) or (154) or (155) or (156) or (157) or (158) or (159) or (160) or (161) or (162) or (163) or (164) or (165) or (166) or (167) or (168) or (169) or (170) or (171) or (172) or (173) or (174) or (175) or (176) or (177) or (178) or (179) or (180) or (181) or (182) or (183) or (184) or (185) or (186) or (187) or (188) or (189) or (190) or (191) or (192) or (193) or (194) or (195) or (196) or (197) or (198) or (199) or (200) or (201) or (202) or (203) or (204) or (205) or (206) or (207) or (208) or (209) or (210) or (211) or (212) or (213) or (214) or (215) or (216) or (217) or (218) or (219) or (220) or (221) or (222) or (223) or (224) or (225) or (226) or (227) or (228) or (229) or (230) or (231) or (232) or (233) or (234) or (235) or (236) or (237) or (238) or (239) or (240) or (241) or (242) or (243) or (244) or (245) or (246) or (247) or (248) or (249) or (250) or (251) or (252) or (253) or (254) or (255) or (256) or (257) or (258) or (259) or (260) or (261) or (262) or (263) or (264) or (265) or (266) or (267) or (268) or (269) or (270) or (271) or (272) or (273) or (274) or (275) or (276) or (277) or (278) or (279) or (280) or (281) or (282) or (283) or (284) or (285) or (286) or (287) or (288) or (289) or (290) or (291) or (292) or (293) or (294) or (295) or (296) or (297) or (298) or (299) or (300) or (301) or (302) or (303) or (304) or (305) or (306) or (307) or (308) or (309) or (310) or (311) or (312) or (313) or (314) or (315) or (316) or (317) or (318) or (319) or (320) or (321) or (322) or (323) or (324) or (325) or (326) or (327) or (328) or (329) or (330) or (331) or (332) or (333) or (334) or (335) or (336) or (337) or (338) or (339) or (340) or (341) or (342) or (343) or (344) or (345) or (346) or (347) or (348) or (349) or (350) or (351) or (352) or (353) or (354) or (355) or (356) or (357) or (358) or (359) or (360) or (361) or (362) or (363) or (364) or (365) or (366) or (367) or (368) or (369) or (370) or (371) or (372) or (373) or (374) or (375) or (376) or (377) or (378) or (379) or (380) or (381) or (382) or (383) or (384) or (385) or (386) or (387) or (388) or (389) or (390) or (391) or (392) or (393) or (394) or (395) or (396) or (397) or (398) or (399) or (400) or (401) or (402) or (403) or (404) or (405) or (406) or (407) or (408) or (409) or (410) or (411) or (412) or (413) or (414) or (415) or (416) or (417) or (418) or (419) or (420) or (421) or (422) or (423) or (424) or (425) or (426) or (427) or (428) or (429) or (430) or (431) or (432) or (433) or (434) or (435) or (436) or (437) or (438) or (439) or (440) or (441) or (442) or (443) or (444) or (445) or (446) or (447) or (448) or (449) or (450) or (451) or (452) or (453) or (454) or (455) or (456) or (457) or (458) or (459) or (460) or (461) or (462) or (463) or (464) or (465) or (466) or (467) or (468) or (469) or (470) or (471) or (472) or (473) or (474) or (475) or (476) or (477) or (478) or (479) or (480) or (481) or (482) or (483) or (484) or (485) or (486) or (487) or (488) or (489) or (490) or (491) or (492) or (493) or (494) or (495) or (496) or (497) or (498) or (499) or (500) or (501) or (502) or (503) or (504) or (505) or (506) or (507) or (508) or (509) or (510) or (511) or (512) or (513) or (514) or (515) or (516) or (517) or (518) or (519) or (520) or (521) or (522) or (523) or (524) or (525) or (526) or (527) or (528) or (529) or (530) or (531) or (532) or (533) or (534) or (535) or (536) or (537) or (538) or (539) or (540) or (541) or (542) or (543) or (544) or (545) or (546) or (547) or (548) or (549) or (550) or (551) or (552) or (553) or (554) or (555) or (556) or (557) or (558) or (559) or (560) or (561) or (562) or (563) or (564) or (565) or (566) or (567) or (568) or (569) or (570) or (571) or (572) or (573) or (574) or (575) or (576) or (577) or (578) or (579) or (580) or (581) or (582) or (583) or (584) or (585) or (586) or (587) or (588) or (589) or (590) or (591) or (592) or (593) or (594) or (595) or (596) or (597) or (598) or (599) or (600) or (601) or (602) or (603) or (604) or (605) or (606) or (607) or (608) or (609) or (610) or (611) or (612) or (613) or (614) or (615) or (616) or (617) or (618) or (619) or (620) or (621) or (622) or (623) or (624) or (625) or (626) or (627) or (628) or (629) or (630) or (631) or (632) or (633) or (634) or (635) or (636) or (637) or (638) or (639) or (640) or (641) or (642) or (643) or (644) or (645) or (646) or (647) or (648) or (649) or (650) or (651) or (652) or (653) or (654) or (655) or (656) or (657) or (658) or (659) or (660) or (661) or (662) or (663) or (664) or (665) or (666) or (667) or (668) or (669) or (670) or (671) or (672) or (673) or (674) or (675) or (676) or (677) or (678) or (679) or (680) or (681) or (682) or (683) or (684) or (685) or (686) or (687) or (688) or (689) or (690) or (691) or (692) or (693) or (694) or (695) or (696) or (697) or (698) or (699) or (700) or (701) or (702) or (703) or (704) or (705) or (706) or (707) or (708) or (709) or (710) or (711) or (712) or (713) or (714) or (715) or (716) or (717) or (718) or (719) or (720) or (721) or (722) or (723) or (724) or (725) or (726) or (727) or (728) or (729) or (730) or (731) or (732) or (733) or (734) or (735) or (736) or (737) or (738) or (739) or (740) or (741) or (742) or (743) or (744) or (745) or (746) or (747) or (748) or (749) or (750) or (751) or (752) or (753) or (754) or (755) or (756) or (757) or (758) or (759) or (760) or (761) or (762) or (763) or (764) or (765) or (766) or (767) or (768) or (769) or (770) or (771) or (772) or (773) or (774) or (775) or (776) or (777) or (778) or (779) or (780) or (781) or (782) or (783) or (784) or (785) or (786) or (787) or (788) or (789) or (790) or (791) or (792) or (793) or (794) or (795) or (796) or (797) or (798) or (799) or (800) or (801) or (802) or (803) or (804) or (805) or (806) or (807) or (808) or (809) or (810) or (811) or (812) or (813) or (814) or (815) or (816) or (817) or (818) or (819) or (820) or (821) or (822) or (823) or (824) or (825) or (826) or (827) or (828) or (829) or (830) or (831) or (832) or (833) or (834) or (835) or (836) or (837) or (838) or (839) or (840) or (841) or (842) or (843) or (844) or (845) or (846) or (847) or (848) or (849) or (850) or (851) or (852) or (853) or (854) or (855) or (856) or (857) or (858) or (859) or (860) or (861) or (862) or (863) or (864) or (865) or (866) or (867) or (868) or (869) or (870) or (871) or (872) or (873) or (874) or (875) or (876) or (877) or (878) or (879) or (880) or (881) or (882) or (883) or (884) or (885) or (886) or (887) or (888) or (889) or (890) or (891) or (892) or (893) or (894) or (895) or (896) or (897) or (898) or (899) or (900) or (901) or (902) or (903) or (904) or (905) or (906) or (907) or (908) or (909) or (910) or (911) or (912) or (913) or (914) or (915) or (916) or (917) or (918) or (919) or (920) or (921) or (922) or (923) or (924) or (925) or (926) or (927) or (928) or (929) or (930) or (931) or (932) or (933) or (934) or (935) or (936) or (937) or (938) or (939) or (940) or (941) or (942) or (943) or (944) or (945) or (946) or (947) or (948) or (949) or (950) or (951) or (952) or (953) or (954) or (955) or (956) or (957) or (958) or (959) or (960) or (961) or (962) or (963) or (964) or (965) or (966) or (967) or (968) or (969) or (970) or (971) or (972) or (973) or (974) or (975) or (976) or (977) or (978) or (979) or (980) or (981) or (982) or (983) or (984) or (985) or (986) or (987) or (988) or (989) or (990) or (991) or (992) or (993) or (994) or (995) or (996) or (997) or (998) or (999) or (1000) or (1001) or (1002) or (1003) or (1004) or (1005) or (1006) or (1007) or (1008) or (1009) or (1010) or (1011) or (1012) or (1013) or (1014) or (1015) or (1016) or (1017) or (1018) or (1019) or (1020) or (1021) or (1022) or (1023) or (1024) or (1025) or (1026) or (1027) or (1028) or (1029) or (1030) or (1031) or (1032) or (1033) or (1034) or (1035) or (1036) or (1037) or (1038) or (1039) or (1040) or (1041) or (1042) or (1043) or (1044) or (1045) or (1046) or (1047) or (1048) or (1049) or (1050) or (1051) or (1052) or (1053) or (1054) or (1055) or (1056) or (1057) or (1058) or (1059) or (1060) or (1061) or (1062) or (1063) or (1064) or (1065) or (1066) or (1067) or (1068) or (1069) or (1070) or (1071) or (1072) or (1073) or (1074) or (1075) or (1076) or (1077) or (1078) or (1079) or (1080) or (1081) or (1082) or (1083) or (1084) or (1085) or (1086) or (1087) or (1088) or (1089) or (1090) or (1091) or (1092) or (1093) or (1094) or (1095) or (1096) or (1097) or (1098) or (1099) or (1100) or (1101) or (1102) or (1103) or (1104) or (1105) or (1106) or (1107) or (1108) or (1109) or (1110) or (1111) or (1112) or (1113) or (1114) or (1115) or (1116) or (1117) or (1118) or (1119) or (1120) or (1121) or (1122) or (1123) or (1124) or (1125) or (1126) or (1127) or (1128) or (1129) or (1130) or (1131) or (1132) or (1133) or (1134) or (1135) or (1136) or (1137) or (1138) or (1139) or (1140) or (1141) or (1142) or (1143) or (1144) or (1145) or (1146) or (1147) or (1148) or (1149) or (1150) or (1151) or (1152) or (1153) or (1154) or (1155) or (1156) or (1157) or (1158) or (1159) or (1160) or (1161) or (1162) or (1163) or (1164) or (1165) or (1166) or (1167) or (1168) or (1169) or (1170) or (1171) or (1172) or (1173) or (1174) or (1175) or (1176) or (1177) or (1178) or (1179) or (1180) or (1181) or (1182) or (1183) or (1184) or (1185) or (1186) or (1187) or (1188) or (1189) or (1190) or (1191) or (1192) or (1193) or (1194) or (1195) or (1196) or (1197) or (1198) or (1199) or (1200) or (1201) or (1202) or (1203) or (1204) or (1205) or (1206) or (1207) or (1208) or (1209) or (1210) or (1211) or (1212) or (1213) or (1214) or (1215) or (1216) or (1217) or (1218) or (1219) or (1220) or (1221) or (1222) or (1223) or (1224) or (1225) or (1226) or (1227) or (1228) or (1229) or (1230) or (1231) or (1232) or (1233) or (1234) or (1235) or (1236) or (1237) or (1238) or (1239) or (1240) or (1241) or (1242) or (1243) or (1244) or (1245) or (1246) or (1247) or (1248) or (1249) or (1250) or (1251) or (1252) or (1253) or (1254) or (1255) or (1256) or (1257) or (1258) or (1259) or (126

## Ada Lovelace's algorithm to compute Bernoulli numbers on Analytical Engine (1843)

## Rare book containing the world's first computer algorithm earns \$125,000 at auction



# An algorithmic success story

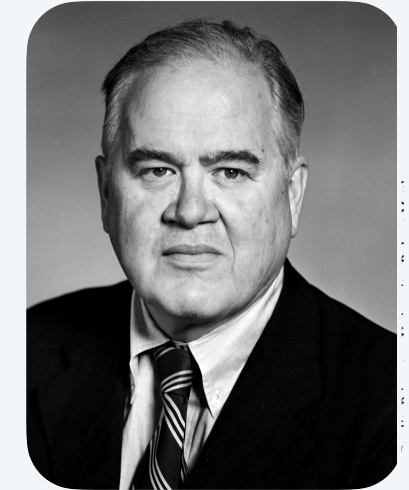
**Goal.** Multiply two polynomials of degree  $n$ .

$$(x^3 + x^2 - 2x + 1) \cdot (3x^3 - x^2 + 2x + 1) = 3x^6 + 2x^5 - 5x^4 + 8x^3 - 4x^2 + 1$$

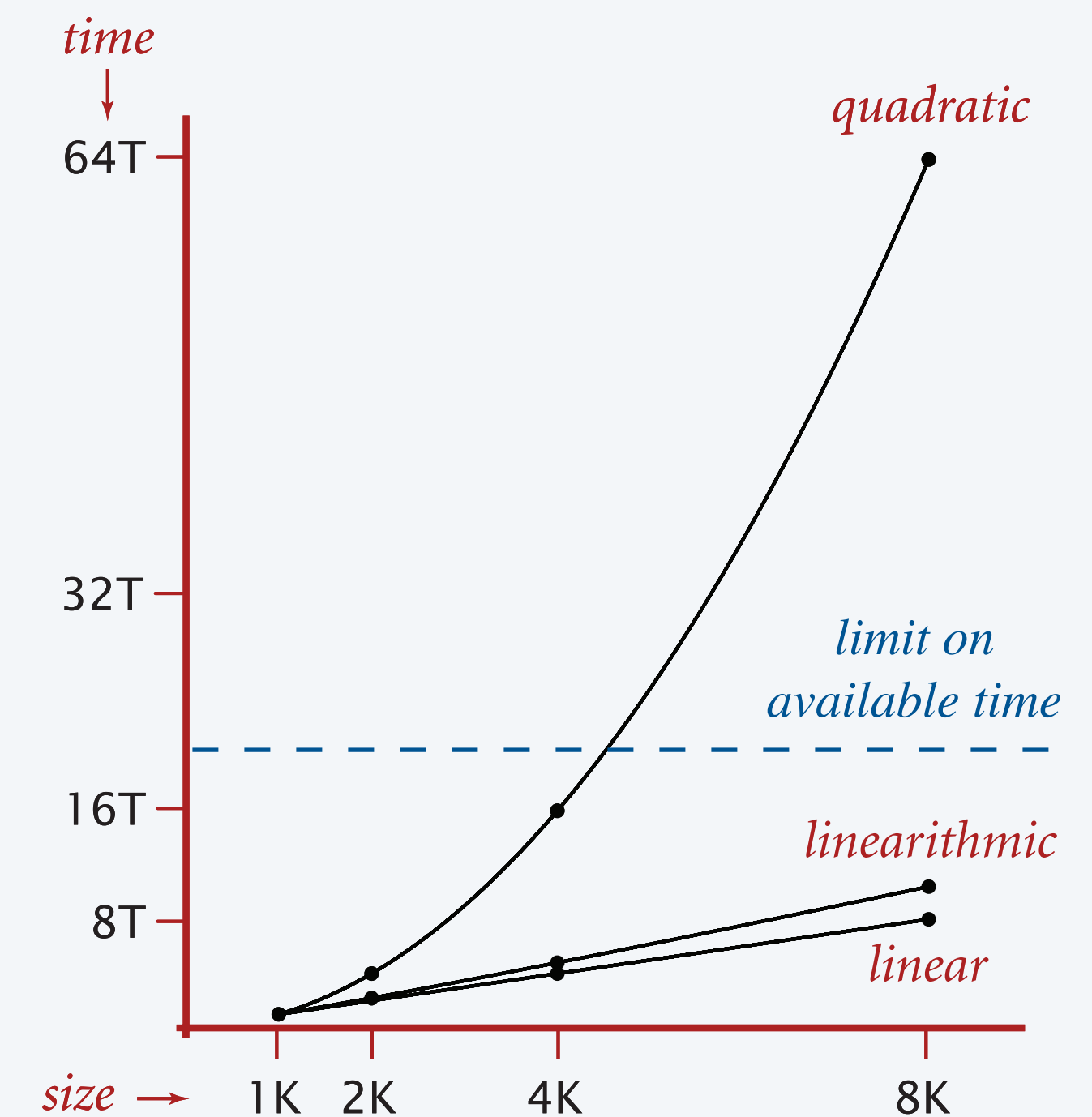
- Applications: JPEG compression, MRI, astrophysics, and more.
- Grade-school algorithm:  $\Theta(n^2)$  operations.
- FFT algorithm:  $\Theta(n \log n)$  operations, **enabling modern technology**.



James  
Cooley



John  
Tukey



# Another algorithmic success story?

## *Why DeepSeek Could Change What Silicon Valley Believes About A.I.*

A new A.I. model, released by a scrappy Chinese upstart, has rocked Silicon Valley and upended several fundamental assumptions about A.I. progress.

Listen to this article • 9:13 min [Learn more](#)

Share full article

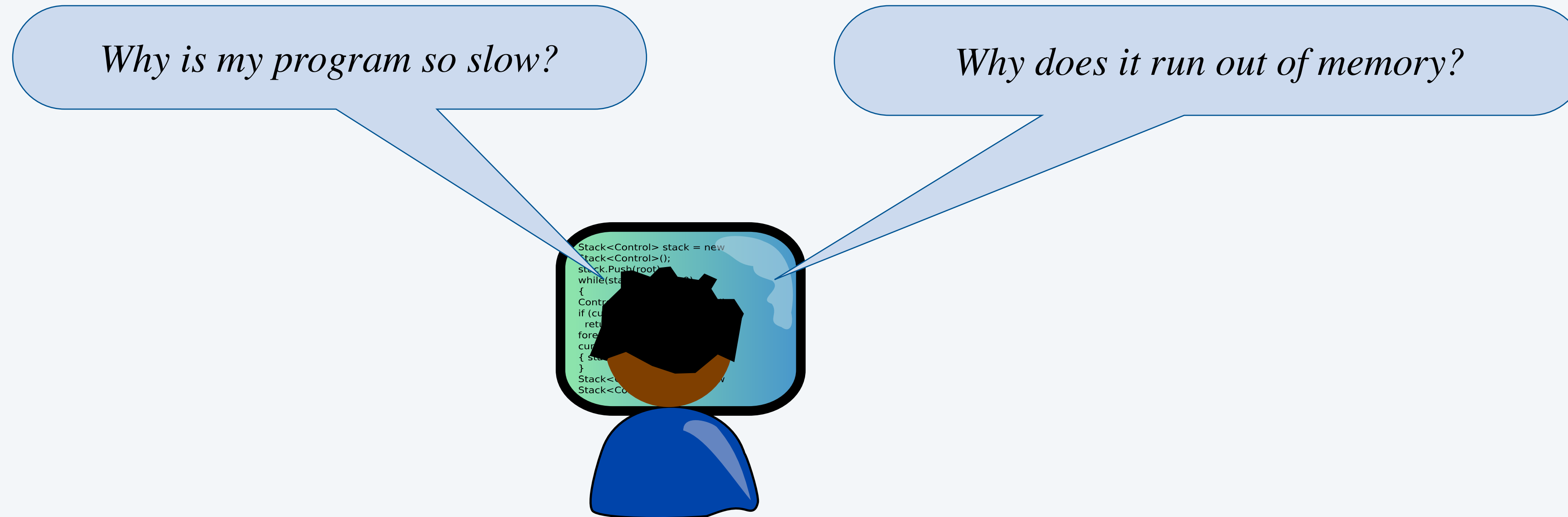
457



# The core challenge

---

- Q1. Will my program handle on large, real-world inputs?  
Q2. If not, how can I analyze and improve its performance?



Our approach: a combination of **experiments** and **mathematical modeling**.

# Example: 3-SUM problem



**Goal.** Given an array of  $n$  distinct integers, count triples  $i < j < k$  such that  $a[i] + a[j] + a[k] = 0$ .

```
~/cos226/analysis> more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

~/cos226/analysis> java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum	
1	30	-40	10	0	✓
2	30	-20	-10	0	✓
3	-40	40	0	0	✓
4	-10	0	10	0	✓

**Context.** Arises in in computational geometry (and even in computer games!)

**Open problem.** What is the optimal running time for solving 3-SUM ?



## 3-SUM problem: brute-force algorithm

---

```
public class ThreeSum {
```

```
    public static int count(int[] a) {
```

```
        int n = a.length;
```

```
        int count = 0;
```

```
        for (int i = 0; i < n; i++)
```

```
            for (int j = i+1; j < n; j++)
```

```
                for (int k = j+1; k < n; k++)
```

```
                    if (a[i] + a[j] + a[k] == 0)
```

```
                        count++;
```

```
        return count;
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        In in = new In(args[0]);
```

```
        int[] a = in.readAllInts();
```

```
        StdOut.println(count(a));
```

```
    }
```

```
}
```

← *count distinct triples that sum to 0*

← *assume no integer overflow*



<https://algs4.cs.princeton.edu>

## 1.4 ANALYSIS OF ALGORITHMS

---

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

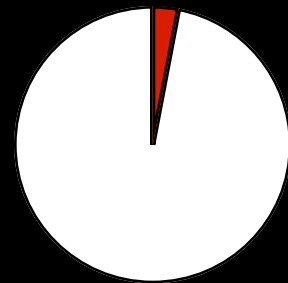


# Measuring running time

**Experiment.** Measure the program's **running time** on inputs of different sizes.

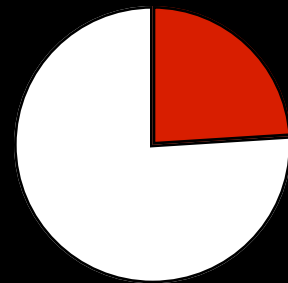
**Observation.** The running time  $T(n)$  increases with the input size  $n$ .

```
~/cos226/analysis> java ThreeSum 1Kints.txt  
70
```



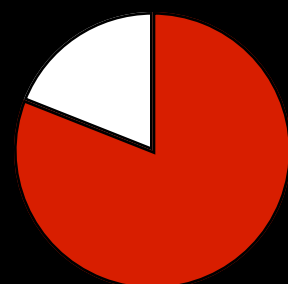
*tick tick*

```
~/cos226/analysis> java ThreeSum 2Kints.txt  
528
```



*tick tick tick tick tick tick tick tick  
tick tick tick tick tick tick tick tick*

```
~/cos226/analysis> java ThreeSum 3Kints.txt  
1670
```



*tick tick tick tick tick tick tick tick tick tick tick tick  
tick tick tick tick tick tick tick tick tick tick tick tick tick  
tick tick tick tick tick tick tick tick tick tick tick tick tick  
tick tick tick tick tick tick tick tick tick tick tick tick tick  
tick tick tick tick tick tick*



# Measuring running time

---

**Experiment.** Measure the program's **running time** on inputs of different sizes.

**Observation.** The running time  $T(n)$  increases with the input size  $n$ .

$n$	time (seconds) †
1,000	0.21
1,500	0.71
2,000	1.63
2,500	3.11
3,000	5.43
4,000	12.8
5,000	25.0
7,500	84.4
10,000	199.3

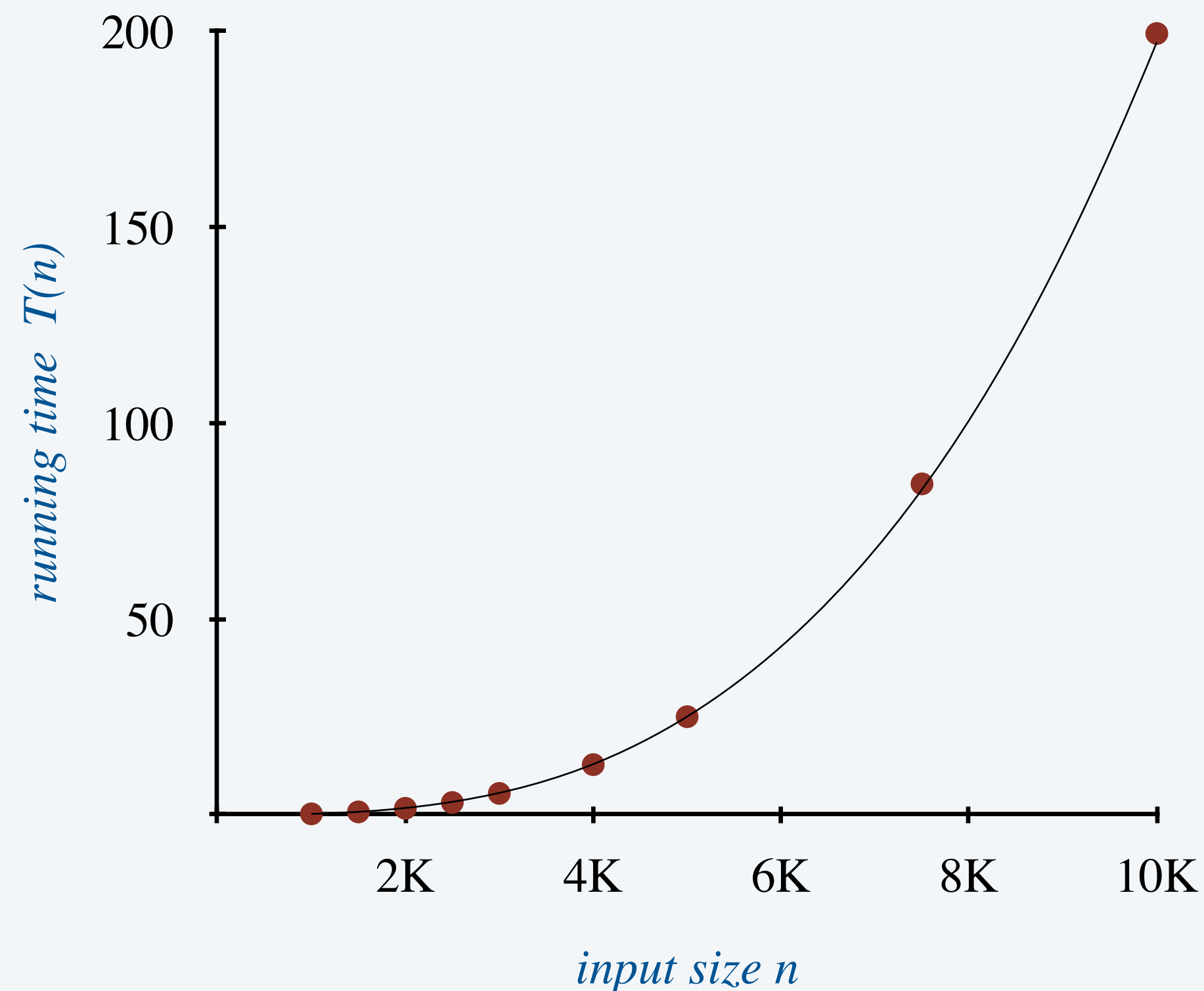


† Apple M2 Pro with 32 GB memory  
running OpenJDK 11 on MacOS Ventura

# Data analysis: running time vs. input size

---

**Visualization.** Plot the running time  $T(n)$  versus the input size  $n$ .



**Hypothesis.** The running time follows a **power law**:  $T(n) = a \times n^b$  seconds.

**Questions.** How can we test this hypothesis? How can we estimate  $a$  and  $b$ ?

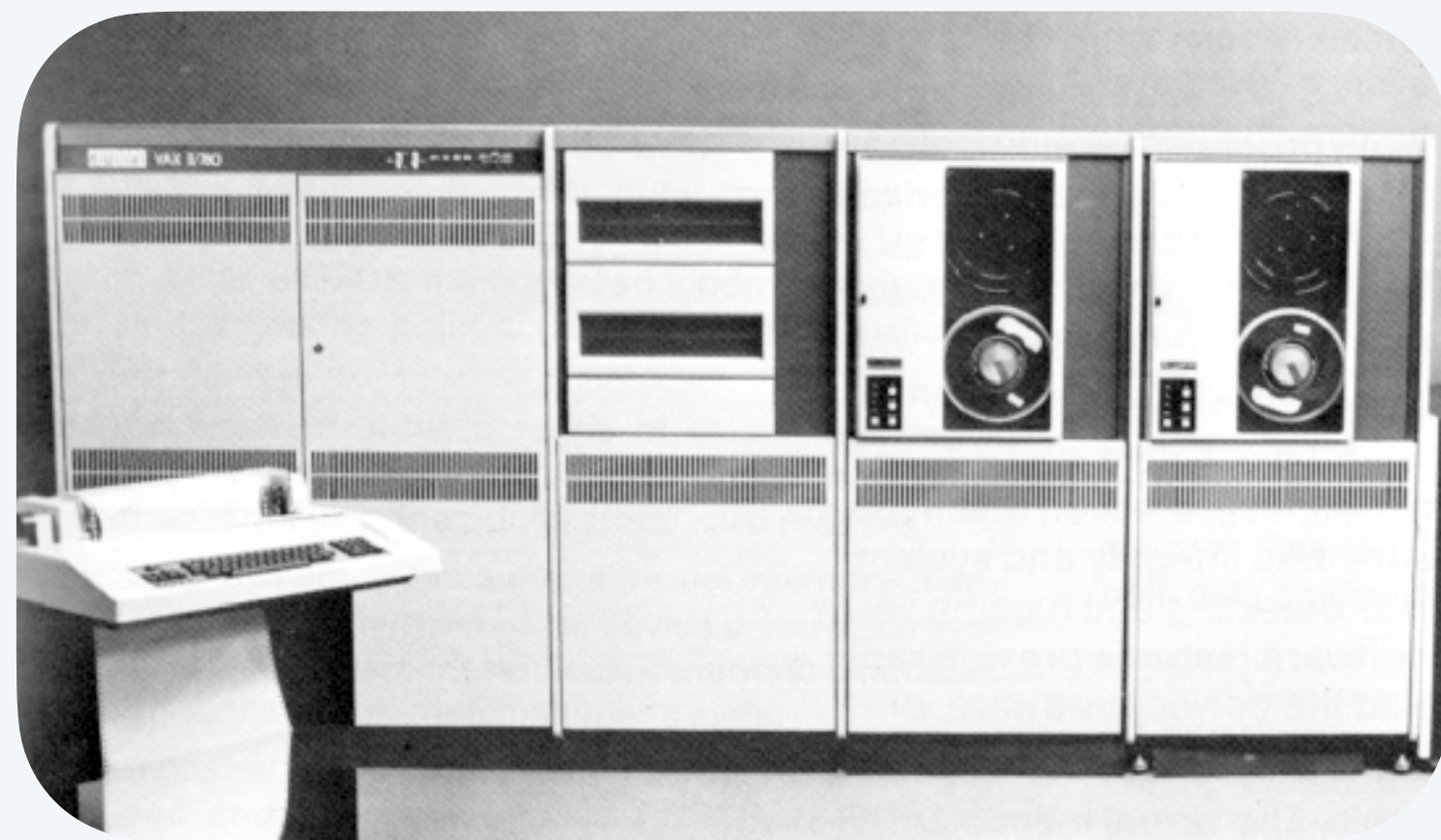
**Answer.** Doubling test,  $\frac{T(n)}{T(n/2)} = 2^b$ .

# Machine invariance

---

**Hypothesis.** For a fixed algorithm, the running times on different computers are the same up to a multiplicative constant factor.

**Note.** That constant factor can be large, sometimes several orders of magnitude.



1970s  
(VAX-11/780)



2020s  
(Macbook Pro M2)

# What affects the running time?

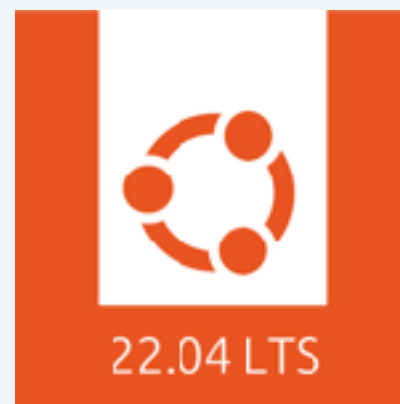
## System independent effects.

- Algorithm.
  - Input data.
- ← *determines exponent  $b$   
in power law  $T(n) = a \times n^b$*

## System dependent effects.

- Hardware: CPU, memory, cache, ...
- Software: compiler, interpreter, garbage collector, ...
- System: operating system, network, other apps, ...

← *determines leading coefficient  $a$   
in power law  $T(n) = a \times n^b$*



Bad news. Getting accurate timing measurements can be difficult. ← *system-dependent effects  
can introduce noise*



<https://algs4.cs.princeton.edu>

## 1.4 ANALYSIS OF ALGORITHMS

---

- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

# Mathematical model of running time

**Model.** The running time =  $\sum$  (frequency of operation)  $\times$  (cost of operation).

- **Frequency of operation:** depends on the algorithm and the specific input.
- **Cost of operation:** depends on hardware, software, system, and low-level implementation details.



**The New York Times**

PROFILES IN SCIENCE

## The Yoda of Silicon Valley

Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, “The Art of Computer Programming.”



**Warning.** For arbitrary programs, frequencies may be impossible to determine.  $\longleftarrow$  *halting problem*

# Example: one-sum problem

Q. How many operations does this code perform as a function of the input size  $n$ ?

```
int count = 0;
for (int i = 0; i < n; i++)
    if (a[i] == 0)
        count++;
```

operation	cost (ns) †	frequency
<i>variable declaration</i>	2 / 5	2
<i>assignment statement</i>	1 / 5	2
<i>less than compare</i>	1 / 5	$n + 1$
<i>equality test</i>	1 / 10	$n$
<i>array read</i>	1 / 10	$n$
<i>increment</i>	1 / 10	$n$ to $2n$

*in practice, depends on  
caching, bounds checking, ...  
(see COS 217)*

*painful to count exactly*

† representative estimates (with a bit of poetic license)

# Simplification 1: cost model

Cost model. Pick one elementary operation as a **proxy** for running time. ← *array accesses, compares, API calls, floating-point operations, ...*

```
int count = 0;
for (int i = 0; i < n; i++)
    if (a[i] == 0) ← “inner loop”
        count++;
```

operation	cost (ns) †	frequency
<i>variable declaration</i>	2 / 5	2
<i>assignment statement</i>	1 / 5	2
<i>less than compare</i>	1 / 5	$n + 1$
<i>equality test</i>	1 / 10	$n$
<i>array read</i>	1 / 10	$n$ ← <i>cost model = array accesses</i>
<i>increment</i>	1 / 10	$n$ to $2 n$

# Simplification 2: asymptotic notation

Tilde notation. Ignore lower-order terms.

Big Theta notation. Ignore both lower-order terms and the leading constant.

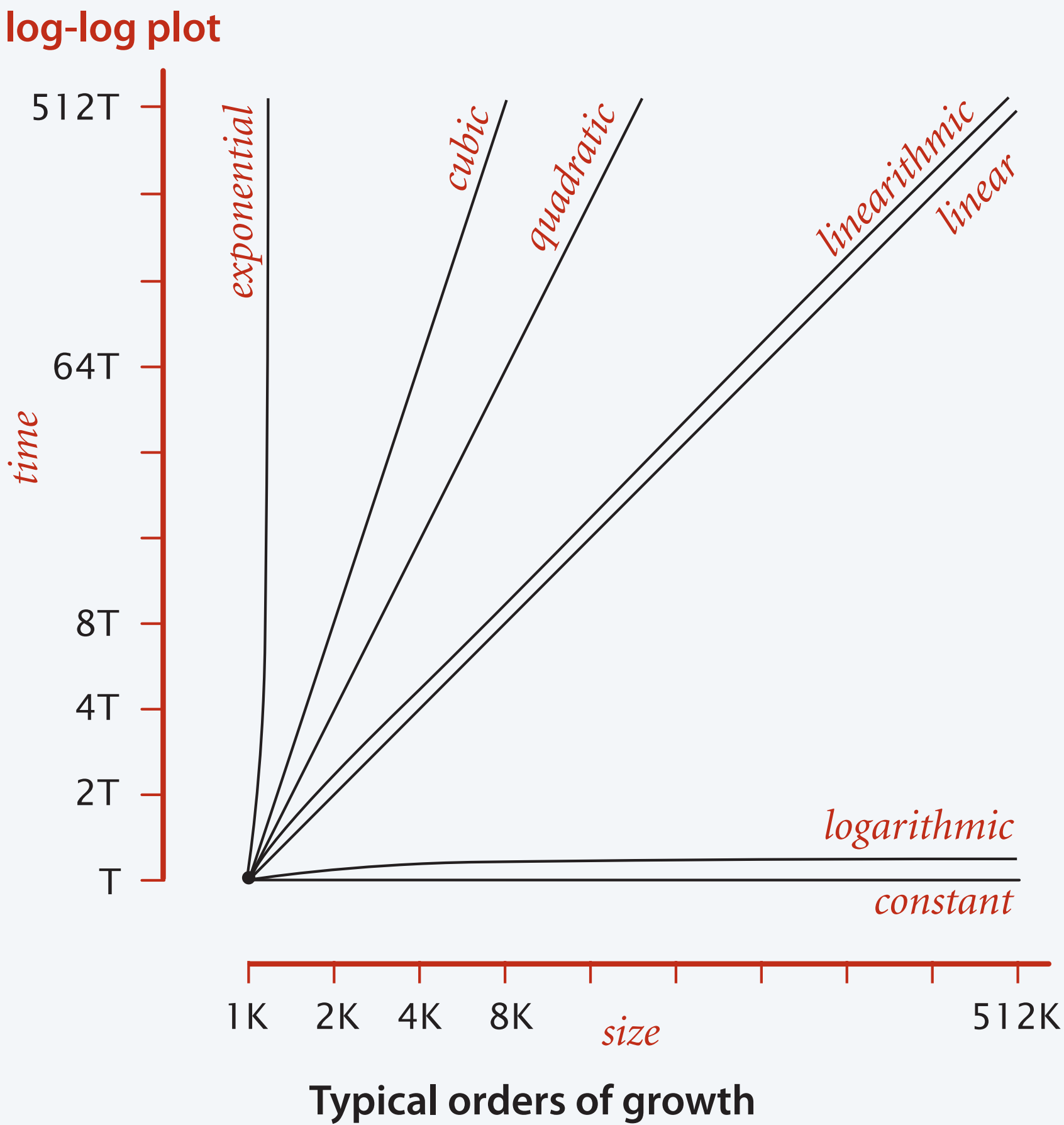
← rigorous definitions involve limits

function	tilde notation	big Theta <span>“order of growth”</span>
$4n^5 + 20n^3 + 16$	$\sim 4n^5$	$\Theta(n^5)$
$0.01n^2 + 10n^{4/3} + 100\log^8 n$	$\sim 0.01n^2$	$\Theta(n^2)$
$2^n + n^5$	$\sim 2^n$	$\Theta(2^n)$
$\underbrace{\frac{1}{6}n^3 - \frac{1}{2}n^2 + \frac{1}{3}n}_{\text{discard lower-order terms}}$	$\sim \frac{1}{6}n^3$	$\Theta(n^3)$

(e.g.,  $n = 1,000$ : 166.667 million vs. 166.167 million)

## Rationale.

- For large  $n$ , lower-order terms have negligible effect.
- For small  $n$ , the value is so small that we don't care.





Which of the following correctly describes the function  $f(n) = n \log_2 n + 3n^2 + 10n$  ?

- A.  $\sim 10n$
- B.  $\sim n \log_2 n$
- C.  $\sim n^2$
- D.  $\Theta(n \log n)$
- E.  $\Theta(n^2)$

## Example: 2-SUM analysis

Q. Approximately how many operations as a function of input size  $n$ ?

```
int count = 0;
for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
        if (a[i] + a[j] == 0)
            count++;
```

← “inner loop”

$$\begin{array}{ccccccccc} i=0 & & i=1 & & i=2 & & & & i=n-2 & & i=n-1 \\ j=1, \dots, n-1 & & j=2, \dots, n-1 & & j=3, \dots, n-1 & & & & j=n-1 & & \text{no } j \\ \downarrow & & \downarrow & & \downarrow & & & & \downarrow & & \downarrow \\ (n-1) & + & (n-2) & + & (n-3) & + & \dots & + & 1 & + & 0 \\ \underbrace{\hspace{15em}} & & & & & & & & & & \\ & & & & & & = & \frac{n(n-1)}{2} \end{array}$$

Step 1. Pick a cost model: array accesses.

Step 2. Count array accesses:  $2 \times \frac{n(n-1)}{2} \sim 1n^2$ .

↑  
*body inner loop makes  
2 array accesses*

Nested loops.

- Independent loops: analyze separately and multiply.
- Dependent loops: write a sum (and simplify).

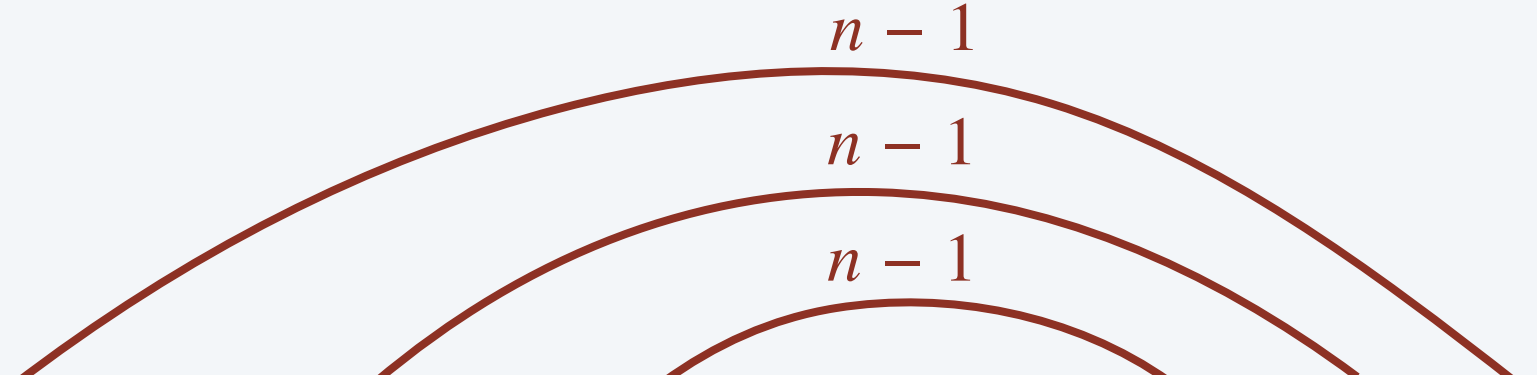
# Triangular sum

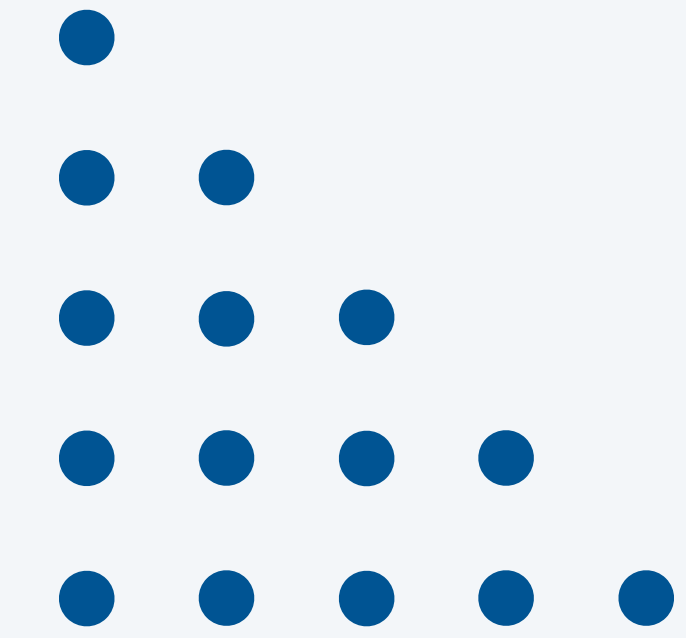
**Claim.**  $0 + 1 + \dots + (n-2) + (n-1) = \frac{1}{2} n(n-1).$

**Proof.**

$$(n-1) + (n-2) + (n-3) + \dots + 2 + 1 + 0 = (n-1) \times (n/2)$$

*sum of each pair*      *number of pairs (assume n is even)*





$$1 + 2 + 3 + \dots + n - 1$$

## Example: 3-SUM analysis

Q. Approximately many operations as a function of input size  $n$ ?

```
int count = 0;
for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
        for (int k = j+1; k < n; k++)
            if (a[i] + a[j] + a[k] == 0)
                count++;
```

← “inner loop”  $\binom{n}{3} = \frac{n(n-1)(n-2)}{3!} \sim \frac{1}{6}n^3$

see COS 240

Step 1. Pick a cost model: array accesses.

Step 2. Count the number of array accesses:  $\Theta(n^3)$ .

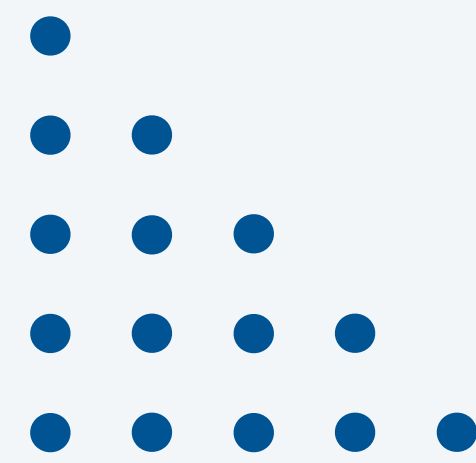
Bottom line. Using a **cost model** and **asymptotic notation** makes analysis manageable.

# Common orders of growth

order of growth	emoji	name	typical code pattern	description	example
$\Theta(1)$	😍	constant	<code>a = b + c;</code>	statement	<i>add two numbers</i>
$\Theta(\log n)$	😎	logarithmic	<code>for (int i = n; i &gt; 0; i /= 2) { ... }</code>	repeatedly divide in half	<i>binary search</i>
$\Theta(n)$	😄	linear	<code>for (int i = 0; i &lt; n; i++) { ... }</code>	single loop	<i>find the maximum</i>
$\Theta(n \log n)$	😁	linearithmic	<i>mergesort</i>	divide-and-conquer	<i>mergesort</i>
$\Theta(n^2)$	😞	quadratic	<code>for (int i = 0; i &lt; n; i++)   for (int j = 0; j &lt; n; j++)   { ... }</code>	double loop	<i>check all pairs</i>
$\Theta(n^3)$	😞	cubic	<code>for (int i = 0; i &lt; n; i++)   for (int j = 0; j &lt; n; j++)     for (int k = 0; k &lt; n; k++)     { ... }</code>	triple loop	<i>check all triples</i>
$\Theta(2^n)$	😈	exponential	<i>towers of Hanoi</i>	brute-force search	<i>check all subsets</i>

# Useful discrete sums and identities

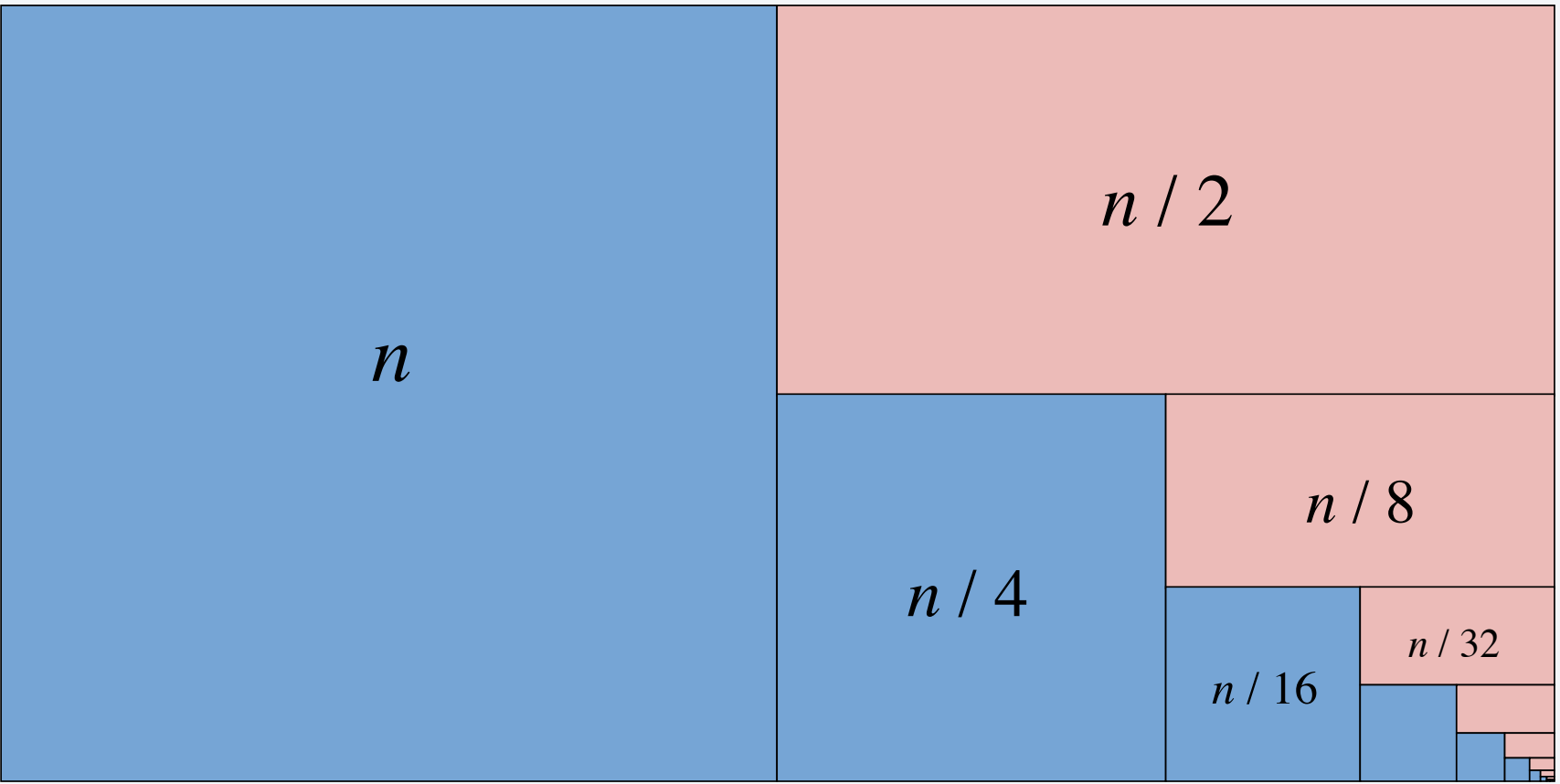
Triangular sum.  $1 + 2 + 3 + \dots + n \sim \frac{1}{2}n^2$



Geometric sum.  $1 + 2 + 4 + 8 + \dots + n = 2n - 1$

Geometric sum'.  $n + \frac{n}{2} + \frac{n}{4} + \dots + 1 = 2n - 1$

*n is a power of 2*



Logarithmic identities.  $\log_2 x + \log_2 y = \log_2(xy)$

$\log_b x = \frac{\log_2 x}{\log_2 b}$  *change of base*



What is the **order of growth** of the running time as a function of  $n$ ?

```
int count = 0;
for (int i = 0; i < n*n; i++)
    for (int j = i+1; j < n*n; j++)
        for (int k = 1; k <= n*n; k = k*2)
            count++;
```

*how would the answer  
change if  $k = k * 4$ ?*

- A.  $\sim \frac{1}{2} n^2 \log_2 n$
- B.  $\sim \frac{1}{2} n^4 \log_2 n$
- C.  $\sim n^4 \log_2 n$
- D.  $\sim \frac{1}{2} n^6$
- E.  $\sim 2 n^6$



What is the **order of growth** of the running time as a function of  $n$ ?

```
int count = 0;
for (int i = n; i >= 1; i = i/2)
    for (int j = 1; j <= i; j++)
        count++;
```

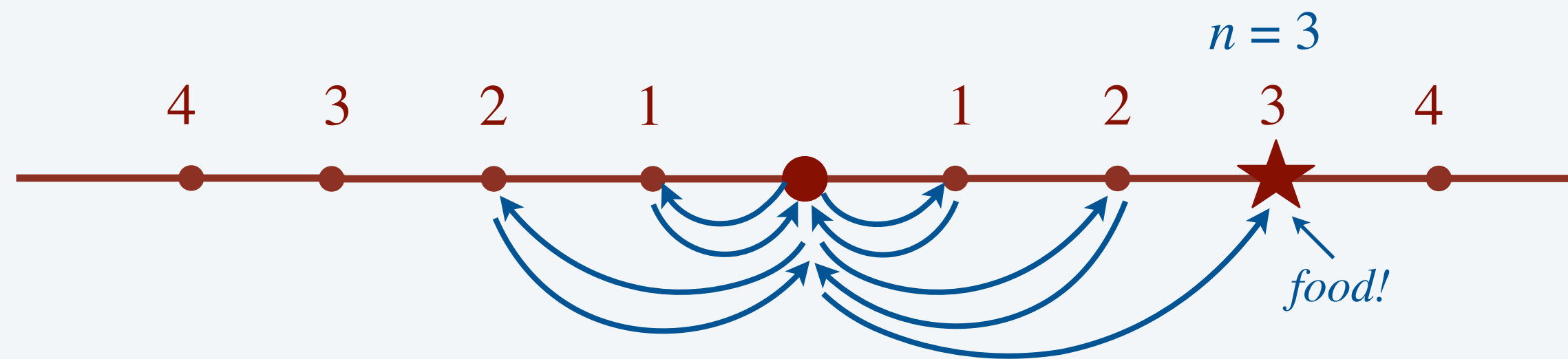
- A.  $\Theta(n)$
- B.  $\Theta(n \log n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(2^n)$

## Example: hungry rat (midterm f22)

---

A rat in a sewer pipe is searching for food. If the nearest food source is  $n$  steps to the right of its starting location, how many steps will it take to reach it using the given strategy?

**Strategy 1:** Take 1 step right, return to start, take 1 step left, return to start.  
Repeat with 2, 3, 4, 5... steps until food found.





A rat in a sewer pipe is searching for food. If the nearest food source is  $n$  steps to the right of its starting location, how many steps will it take to reach it using the given strategy?

*assume  $n$  is a power of 2*

**Strategy 2:** Take 1 step right, return to start, take 1 step left, return to start.

Repeat with 2, 4, 8, 16... steps until food found.

- A.  $\Theta(\log n)$
- B.  $\Theta(n)$
- C.  $\Theta(n \log n)$
- D.  $\Theta(n^2)$
- E.  $\Theta(2^n)$



<https://algs4.cs.princeton.edu>



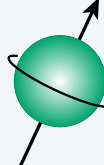



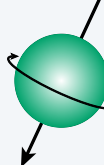

## 1.4 ANALYSIS OF ALGORITHMS

---

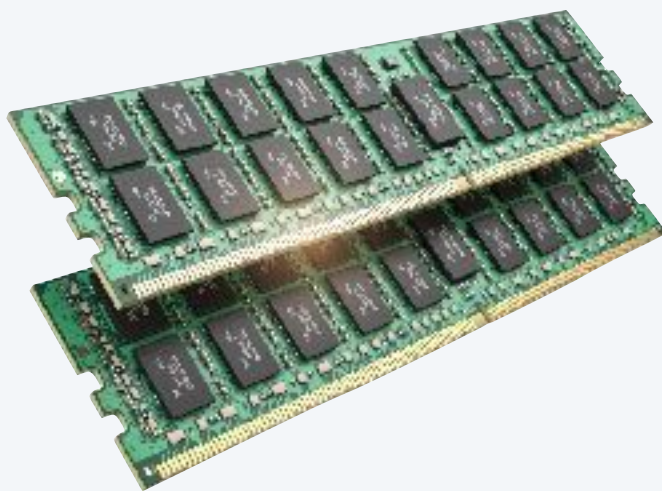
- *introduction*
- *running time (experimental analysis)*
- *running time (mathematical models)*
- *memory usage*

# Memory basics: bits, bytes, and pointers

Bit. A single binary digits (0 or 1).

0				
1				

term	symbol	size
byte	B	8 bits
kilobyte	KB	$10^3$ bytes
megabyte	MB	$10^6$ bytes
gigabyte	GB	$10^9$ bytes
terabyte	TB	$10^{12}$ bytes



↑  
*some systems use powers of 2  
(e.g., 1 MB =  $2^{20}$  bytes)*

Assumption. Running on a 64-bit machine with 8-byte pointers.



↑  
*some JVMs “compress” pointers  
to 4 bytes to avoid this cost*

# Typical memory usage of primitive types and arrays in Java

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8
primitive types	

type	bytes
boolean[]	$\sim 1\ n$
int[]	$\sim 4\ n$
double[]	$\sim 8\ n$
one-dimensional arrays (length n)	

← array overhead = 24 bytes

type	bytes
boolean[][]	$\sim 1\ n^2$
int[][]	$\sim 4\ n^2$
double[][]	$\sim 8\ n^2$
two-dimensional arrays (n-by-n array of arrays)	

type	bytes
object reference	8

↑  
64-bit machine

# Typical memory usage for objects in Java

**Objects memory** = sum of memory for instance variables + overheads

**Ex.** Each *Date* object uses 32 bytes of memory.

```
public class Date {  
    private int day;  
    private int month;  
    private int year;  
  
    ...  
}
```



16 bytes (object overhead) ← *class pointer, garbage collector bits, lock state, hash code, ...*

4 bytes (int)

4 bytes (int)

4 bytes (int)

4 bytes (padding, round to a multiply of 8 bytes)

---

32 bytes

Array declaration is 8 bytes.

`Date[] dates;` ← *reference*

When *dates* contains  $n$  elements, it uses  $\Theta(n)$  bytes.



How much memory does a `WeightedQuickUnionUF` object use as a function of  $n$ ?

- A.  $\sim 4n$  bytes
- B.  $\sim 8n$  bytes
- C.  $\sim 4n^2$  bytes
- D.  $\sim 8n^2$  bytes

```
public class WeightedQuickUnionUF {  
    private int[] parent;  
    private int[] size;  
    private int count;  
  
    public WeightedQuickUnionUF(int n) {  
        parent = new int[n];  
        size = new int[n];  
  
        count = 0;  
        for (int i = 0; i < n; i++)  
            parent[i] = i;  
        for (int i = 0; i < n; i++)  
            size[i] = 1;  
    }  
    ...  
}
```

# Credits

---

image	source	license
<i>Charles Babbage</i>	<u>The Illustrated London News</u>	<u>public domain</u>
<i>Babbage Engine in Operation</i>	<u>xRez Studio</u>	
<i>Algorithm for the Analytical Engine</i>	<u>Ada Lovelace</u>	<u>public domain</u>
<i>Ada Lovelace and Book</i>	<u>Moore Allen &amp; Innocent</u>	
<i>Galaxies Colliding</i>	<u>SaltyMikan</u>	
<i>James Cooley</i>	<u>IEEE</u>	
<i>John Tukey</i>	<u>Princeton University</u>	
<i>Programmer Icon</i>	<u>Jaime Botero</u>	<u>public domain</u>
<i>Head in the Clouds</i>	<u>Ellis Nadler</u>	<u>education</u>
<i>Student Raising Hand</i>	<u>classroomclipart.com</u>	<u>educational use</u>
<i>Running Time</i>	pano.si	
<i>Analog Stopwatch</i>	<u>Adobe Stock</u>	<u>education license</u>

# Credits

---

image	source	license
<i>Analog Stopwatch</i>	<u>Adobe Stock</u>	<u>education license</u>
<i>Apple M4 Chip</i>	<u>Apple</u>	
<i>Macbook Pro M2</i>	<u>Apple</u>	
<i>Scientific Method</i>	<u>Sue Cahalane</u>	by author
<i>Laboratory Apparatus</i>	<u>pixabay.com</u>	<u>public domain</u>
<i>Dissected Rat</i>	<u>Allen Lew</u>	<u>CC BY 2.0</u>
<i>Harmonic Integral</i>	<u>Wikimedia</u>	<u>public domain</u>
<i>Geometric Series</i>	<u>Wikimedia</u>	<u>CC BY-SA 3.0</u>
<i>Recursive Load</i>	<u>Marek Bennett</u>	
<i>The Yoda of Silicon Valley</i>	<u>New York Times</u>	
<i>Babbage's Analytical Engine</i>	<u>Science Museum, London</u>	<u>CC BY-SA 2.0</u>
<i>Alan Turing</i>	<u>Science Museum, London</u>	

## A final thought

---

*“It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then give them various weights.”* — Alan Turing (1947)

